21世纪高职高专规划教材

# Visual C++程序设计

王明福　主编
余苏宁　副主编

高等教育出版社

21 世纪高职高专规划教材

# Visual C++ 程序设计

王明福　主　编
余苏宁　副主编

高等教育出版社

## 内 容 提 要

本书从实际应用的角度介绍了 Visual C++6.0软件包的使用方法和编程技巧。通过开发计算器、学生档案管理程序、绘图程序、多媒体点播系统、公众聊天室和桌面时差时钟等程序，详细介绍了包括菜单、对话框、常用控件、工具栏等在内的界面设计，MFC 库的使用和扩展以及对文件、多媒体、数据库、网络通信和多线程等编程技术的具体操作技巧。

本书改变传统写法，采用“项目”驱动的编写方式，把知识点融入到实际项目的开发中，通过项目的不断扩展逐步引入新的知识点，通俗易懂，可操作性强。适合高等院校“Visual C++ 程序设计”课程教学用书，尤其对高职高专院校计算机专业和从事 Visual C++ 的编程开发人员，更是一本很难得的好书。

书中所有程序全部运行通过，所有程序源代码及示例相关文档均可以从高等教育出版社网站上下载，网址为：http://cs.hep.com.cn 或 http://www.hep.edu.cn。

# 前　言

Microsoft Visual C++6.0 是微软公司最新推出的开发工具包 Visual Studio 6.0 中功能最强大的开发工具，是一种面向对象程序设计语言，可以大大提高软件设计能力以及开发速度。基于对当前教材的深入了解及教学实践需要，我们着手编写了 21 世纪高职高专规划教材《Visual C++ 程序设计》。本书从实际应用的角度介绍了 Visual C++6.0软件包的使用方法和编程技巧。通过开发计算器、学生档案管理程序、绘图程序、多媒体点播系统和公众聊天室等程序，详细介绍了包括菜单、对话框、常用控件、工具栏等在内的界面设计方法、MFC 库的使用和扩展以及对文件、多媒体、数据库、网络通信和多线程等编程技术的具体操作使用技巧。

本书改变传统教材的编写方法，具有如下特点：

1. 采用“项目”驱动的编写方式，引入案例教学和启发式教学方法，便于激发学习兴趣。

本书的编写思想是把知识点融入到实际项目的开发中去，立足于“理论够用，操作熟练，重在实践”的要求，力求做到通俗易懂，循序渐进，适合以“案例入门，改造拓宽，项目综合”的学习知识体系模式展开教学。每一章都尽量从实际中的典型实例开始，引出问题，由问题牵引，然后逐步展开，在讲述实例或开发项目的过程中，将本章的知识融入。通过对问题的深化或功能扩充，来拓宽知识的广度和深度，最后达到学习知识、培养能力的目的。这种将知识点融入到实际项目开发中的编写方式，可读性、可操作性强，非常适合高职高专的学生阅读和使用。

2. 在案例或项目的选择上，遵循“易学”、“有趣”和“有用”的原则，这样有利于激发学生的求知欲望。

本教材所选案例（或项目）基本包含了“Visual C++ 程序设计”的基本概念和设计技巧，由浅入深、循序渐进、逐步拓宽知识点。兼顾了理论知识的系统性和完整性，考虑到了独立和相关的平衡，其总目标是强调综合应用开发能力的培养。换言之，既能实践循序渐进的教学方法，也有利于开展“项目综合”的教学模式，符合教学规律。

3. 一切以实用为目的，注重知识应用的先进性和前沿性，具有高职教育特色。

本教材着眼于 IT 产业飞速发展的需要，将多媒体编程技术、数据库技术和网络通信技术纳入教材内容。本书不追求面面俱到，而是大胆舍去不用或根本就不实用的内容，适合“理论够用，重在实践”的高职高专教学的特点。

4. 本书注重 C 语言程序设计系列的个性和共性，考虑到两个方面的平滑过渡及其中的异同点。一是从面向过程的程序设计到面向对象程序设计的平滑过渡及异同点，二是在 DOS 环境下与 Windows 环境下程序的平滑过渡及异同点。反映在章节内容的安排上，第 1 章介绍了 Visual C++ 编程基础，其目的就是让读者了解 DOS 程序与 Windows 程序的差别以及 MFC 应用程序的结构；第 2 章介绍 MFC 的界面设计与资源管理，其目的是让读者掌握用 VC 开发平台编写 MFC 应用程序的一些基本操作；从第 3 章开始才是本教材的主要内容。

本书由深圳职业技术学院计算机系王明福、余苏宁两位老师编写。其中第 2 章由余苏宁老

师编写，其余各章由王明福老师编写，最后由王明福负责全书的统稿。此外，李亮、鸟云高娃和王梅等老师在百忙之中对本书的编写思路提出了许多宝贵意见，并承担了部分章节的文字录入与审校工作；同时，本书也得到了计算机系软件专业全体老师的大力支持，提出了许多建设性意见。在此，我们深表感谢。

由于编者水平有限，加之时间仓促，书中难免有错漏之处，敬请广大读者批评指正，在此表示感谢。编者 E－mail 地址是 wmf@oa.szpt.net。

编 者

2003 年 5 月于深圳

# 目　录

# 第 1 章

# Visual C++ 编程基础

## 本章导读

Windows 下编程的工具多种多样，但均遵循 Windows 的工作原理。作为一名程序员，必须熟练掌握与编程有关的 Windows 的工作原理。选择能为用户提供一种功能强大同时又易于掌握的应用程序开发工具。

本章通过介绍 Windows 编程的基础知识，编写一句代码的 MFC（Microsoft Foundation Class）应用程序，让读者了解：

- Windows 编程基础：消息驱动、资源管理等
- MFC 类基础
- 利用向导建立一个应用程序框架

---

## 1.1 Windows 编程概念

传统的 MS－DOS 程序主要采用顺序的、关联的、过程驱动的程序设计方法，是面向程序而不是面向用户的。Windows 程序设计是基于事件驱动的，程序的运行不是由事件的顺序来控制，而是由能触发的事件来控制，它是一种面向用户的程序设计方法。其中消息驱动机制是 Windows 程序设计的精髓。

### 1.1.1 事件与消息

Windows 花费大量时间等待用户的动作以便做出响应，所以这种系统也叫事件驱动的系统。例如，当用户按下一个键、移动鼠标或单击鼠标按钮时，计算机通知 Windows 系统已经发生了一个事件以及事件的种类、发生的时间和发生的位置（如坐标值）。

事件以如下三种方式产生：

（1）通过输入设备，如键盘和鼠标，产生硬件事件。

（2）通过屏幕上可视的对象，如菜单、工具栏按钮、滚动条和对话框上的控件。

（3）来自 Windows 内部，例如，让一个后面的窗口显示到前面来。

当 Windows 捕获一条事件后，它会编写一条消息，将相关信息放入一个数据结构中，然后将包含此数据结构的消息发送给需要消息的程序。Windows 消息是在 Windows.h 文件中用宏定义的常数。消息常数名通常为 WM _ XXX，例如，WM _ WUIT、WM _ CHAR 等。

Windows 将消息放入目标应用程序的消息队列中，在消息队列中所有消息都处于等待状态，直到应用程序准备处理它。

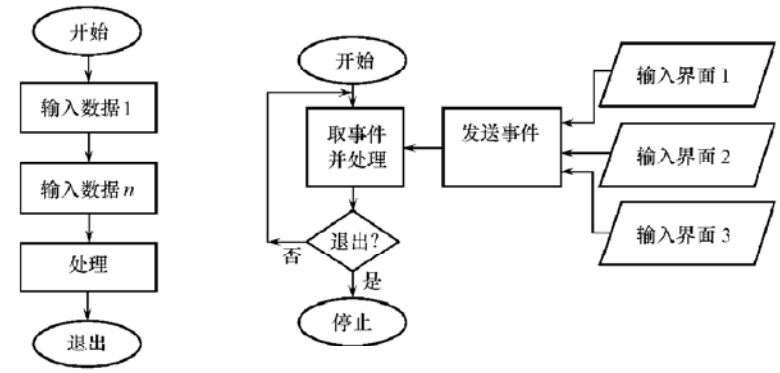
### 1.1.2　消息驱动

消息驱动也称事件驱动，Windows 程序设计是基于事件驱动的，Windows 对消息有一套完善严格的定义，并在其产生时将其发送给所有相关的应用程序，这些消息用于驱动应用程序运行以实现一定的功能。

例如，当用户单击鼠标左键时，将发送 WM _ LBUTTONDOWN 消息；而双击时，则发送 WM _ LBUTTONDBLCLK 消息。除提供的标准消息外，程序员可以自定义所需的消息。总之，消息驱动是 Windows 程序设计的精髓。

传统的 MS－DOS 程序是一系列预先定义好的操作序列的组合，具有一定的开头、中间过程和结束，也就是程序直接控制程序事件和过程的顺序。它的基本模型如图 1.1(a)所示。

事件驱动的程序设计不是由事件的顺序来控制，而是由事件的发生来控制，而这种事件的发生是随机的、不确定的，并没有预定的顺序，这样就允许程序的用户用各种合理的顺序来安排程序的流程。它是一种面向用户的程序设计方法，它在程序设计过程中除了完成所需功能之外，更多地考虑了用户的各种输入，并根据需要设计相应的处理程序。它是一种被动式的程序设计方法，程序开始运行时，处于等待用户输入事件状态，然后取得事件并做出相应反应，处理完毕又返回并处于等待事件状态，如图 1.1(b)所示。




(a) 过程驱动程序模型　　(b) 事件驱动程序模型

图 1.1　过程驱动与事件驱动模型之比较

### 1.1.3　消息响应函数

消息响应函数是用于处理特定消息的一些代码。收到消息的应用程序会做些什么，取决于应用程序本身。程序员可以编写相应的处理函数以处理消息。例如，当用户点击某菜单项时，希望程序弹出一个口令对话框，那么，只要在相应的消息处理函数中编写弹出一个口令对话框的代

码即可。

对于特定的消息有许多标准或典型的处理。例如，WM _ PAINT 消息(在窗口重新绘制内容时发送)的处理函数需要采取步骤，重新构造显示在窗口的图像，重新绘制可见的文本、图形，等等。

### 1.1.4 资源管理

程序员设计任何应用程序均将涉及到诸如菜单、对话框、消息框以及按钮等标准格式数据。Windows 将这些数据保存在资源文件中，程序员可通过编辑工具编辑、修改这些资源文件，使其提供所需的菜单或按钮，并将其放入设计的程序之中。

DOS 下的内存管理常常是程序设计的瓶颈问题，而在 Windows 下编程时，只需要简单地申请所需内存即可。

### 1.1.5 设备独立性

设备独立性或称设备无关性，是 Windows 95/98/2000 的突出特点之一。在编程时，程序员不必关心诸如打印机、鼠标、键盘、显示器、声卡、显示卡和 CDROM 等多种设备的类型及其驱动程序。Windows 95/98/2000 提供了图形设备接口之类的各种抽象接口，使得在程序中可以通过调用标准函数与硬件交互，这些函数通过设备环境的数据结构，由 Windows 操作系统将其映射到相应的物理设备，而程序员则无需了解其提供的操作设备的各种指令。

## 1.2 MFC 基础

MFC 应用程序框架生成的应用程序使用了标准的结构，很好地解决了因程序结构不同，导致维护程序和程序员之间交流的障碍。

Visual C++ 已集成了 MFC 库、Visual C++ 资源编辑器、AppWizard 和 ClassWizard，明显减少了应用程序编码的时间。AppWizard 为整个应用程序生成框架代码，同时 ClassWizard 为消息处理程序生成原型和函数体。

在本节先介绍 MFC 类库，然后介绍程序框架中的一些要素，接下来介绍 MFC 消息映射，最后介绍程序的运行过程，使读者对 MFC 有一个大概的了解。

### 1.2.1 MFC 类库简介

MFC(Microsoft Foundation Class)类库是用来编写 Windows 应用程序的 C++ 类集，封装了大部分 Windows API 函数，使用 MFC 类库和 Visual C++ 提供的高度可视化的应用程序开发工具，可使应用程序的开发变得简单，而可靠性和可重用性得到很大提高。

MFC 库包括 CObject 类及其派生类以及其他类(可以参考 MSDN 来了解 MFC 类库层次图)。了解了 CObject 类及其派生类，就等于了解了 MFC 的一大半。

在 MFC 类库中，有这样一些重要的类(都是 CObject 的派生类)：

(1) 应用程序类 CWinApp　属于 Application Architechture(应用程序体系结构)。一个 MFC 项

目对应一个此类的对象。

（2）CWnd 类及派生类　属于 Windows Support 部分，用户看到的 Windows 界面都是由这个类的对象所形成。它包括这样几部分：

① Frame Windows　包括用于生成框架窗口的 CFrameWnd 及其派生类，用于生成分隔栏窗口的 CSplitterWnd。

② Views　包括 CView 及其派生类，用于生成视图窗口。

③ Dialog　包括 CDialog 及其派生类，用于生成对话框。

④ Control Bars　包括 CControlBar 及其派生类，用于生成状态栏和工具栏等。

⑤ Property Sheets　包括 CPropertySheet 及其派生类，用于生成属性表。

⑥ Controls　包括各种控件类，比如，CEdit 用于生成编辑框，CListBox 用于生成列表框等。

（3）CDocument 及其派生类　和 CWinApp 同属于 Application Architechture 范畴。用于提供应用程序数据的存储和加载，常和 CView 类一起工作，合在一起称为文档/视图结构。

（4）File Services　包括 CFile 各类，提供文件服务，这是一种底层的服务，一般可以用一些高层的服务代替（比如文档的序列化等）。

（5）Graphical Drawing　包括 CDC（Class of Device Context，设备环境类）等类，提供图形绘制功能。当希望在视图窗口中绘图时，需要使用该 CDC，以提供绘图环境。

（6）Graphical Drawing Objects　绘图环境，可以提供绘图对象，比如画笔（CPen）、刷子（CBrush）等来进行绘制。

（7）Menus　包括 CMenu 类，封装了 Windows 中菜单的数据结构。

（8）ODBC Database 和 DAO Database Support　包括 CDatabase 和 CDaoDatabase 等类，提供数据库服务。

（9）Internet Services　包括 CInternetSession 等各类，提供网络服务。

此外，还有一些非 CObject 类及派生类，也是必须了解的。比如：

（10）Simple Value Types　包括 CPoint、CRect、CSize、CString、CTime 和 CTimeSpan。

（11）Internet Service API　包括 CHttpServer 等各类，用于提供底层的网络服务，对这些类的理解最好能结合实例进行。

### 1.2.2　MFC 应用程序框架

应用程序框架包含用于生成应用程序所必需的各种面向对象的组件的集合。在 Visual C++ 6.0 中，MFC AppWizard 能方便地生成应用程序框架，然后可以在此基础上进行进一步的编辑工作。MFC AppWizard 生成的应用程序包括如下一些要素：

（1）WinMain()函数　Windows 要求应用程序必须有一个 WinMain()函数。但在程序中看不到 WinMain()，因为它隐藏在应用程序框架中。

（2）应用程序类　也称 CMyHelloApp。该类的每一个对象代表一个应用程序。程序中默认定义一个全局 CMyHelloApp 对象，即 theApp。CWinApp 基类决定 theApp 的大多数行为。

（3）应用程序启动　启动应用程序时，Windows 调用应用程序框架内置的 WinMain()函数，WinMain()寻找由 CWinApp 派生出的全局构造的应用程序对象。在 C++ 程序中，全局对象在主程序执行之前构造。

(4) 成员函数 CMyHelloApp::InitInstance() 当 WinMain()函数找到应用程序对象时,它调用伪成员函数 InitInstance(),这个成员函数调用所需的构造并显示应用程序的主框架窗口。必须在派生的应用程序类中重载 InitInstance(),因为 CWinApp 基类不知道需要什么样的主框架窗口。

(5) 成员函数 CWinApp::Run()。函数 Run()隐藏在基类中,但是它发送应用程序的消息到窗口,以保持应用程序的正常运行。在 WinMain()调用 InitInstance()之后,便调用 Run()。

(6) CMainFrame 类 类 CMainFrame 的对象代表应用程序的主框架窗口。当构造函数调用基类 CFrameWnd 的成员函数 Create()时,Windows 创建实际窗口结构,应用程序框架把它连接到 C++ 对象,函数 ShowWindows()和 UpdateWindow()也是基类的成员函数,必须调用它们来显示窗口。

(7) 文档与视图类 这一部分比较复杂,会在后面的章节中单独提出详细讲解。

(8) 关闭应用程序 如果用户通过关闭主框架窗口类关闭应用程序,这个操作就将激发一系列事件的发生,包括 CMainFrame 对象的析构、从 Run()中退出、从 WinMain()中退出和 CMyHelloApp 对象的析构。

### 1.2.3 MFC 消息映射及处理

在 MFC 中,管理消息的方式通常是这样的:当发生某一个消息(比如用户移动了鼠标和按下键盘等),该消息进入消息队列,操作系统根据消息提供的信息决定由哪个应用程序来处理,该应用程序依照一定的方式查找应用程序中各个类的消息映射(一组宏,这些宏用来确定某个消息及相应的处理程序的对应关系),找到处理程序,然后由处理程序执行。

MFC 程序要处理消息的种类如下:

(1) Windows 消息 这类消息以 WM_开头,但 WM_COMMAND 除外。通常由窗口和视图来处理。这些消息常常带有参数,用于决定处理该消息的方式。

(2) 由控件和其他子窗口发送给父窗口的 WM_COMMAND 消息 这些消息中包括 EN_CHANGE 通知码。例如,当用户在编辑框中键入文本或进行修改时,就会向系统发送一个带 EN_CHANGE 通知码的 WM_COMMAND 消息。

(3) 来自于用户界面对象的 WM_COMMAND 消息 如表 1.1 所示。这些用户界面对象包括菜单、工具栏按钮和快捷键。系统处理这些用户界面对象消息的方式和处理前面的消息有所不同,当这种类型对象接受某个消息时,它将处理该消息的权利优先提供给其他对象。

表 **1.1** 用户界面对象发送消息处理顺序表

| 对 象 | 处理次序 | |
|---|---|---|
| MDI 框架窗口<br>(CMDIFrameWnd) | 1. 活动的 CMDIChildWnd<br>3. 应用程序(CWinApp 对象) | 2. 本框架窗口 |
| 文档窗口<br>(CFrameWnd, CMDIChildWnd) | 1. 活动视图<br>3. 应用程序(CWinApp 对象) | 2. 本框架窗口 |
| 视图 | 1. 本视图 | 2. 和视图相关的文档 |
| 文档 | 1. 本文档 | 2. 和该文档相关的文档模板 |
| 对话框 | 1. 本对话框<br>3. 应用程序(CWinApp 对象) | 2. 负责该对话框的窗口 |

### 1.2.4　程序的运行过程

如图 1.2 所示，当运行用户应用程序时，程序中的框架首先获得控制权，然后依次执行下述功能：

（1）做部分初始化工作。

（2）构造应用程序的惟一应用类的对象，构造应用类对象时要调用其构造函数。

（3）调用 WinMain( )函数（此函数也隐藏在应用框架内部）。注意，调用函数 WinMain( )时已将其 hPrevInstance 参数强行置为 NULL，这一点与 SDK 有区别。

（4）从 WinMain( )函数返回后，删除应用程序的惟一应用类的对象，删除时要调用其析构函数。

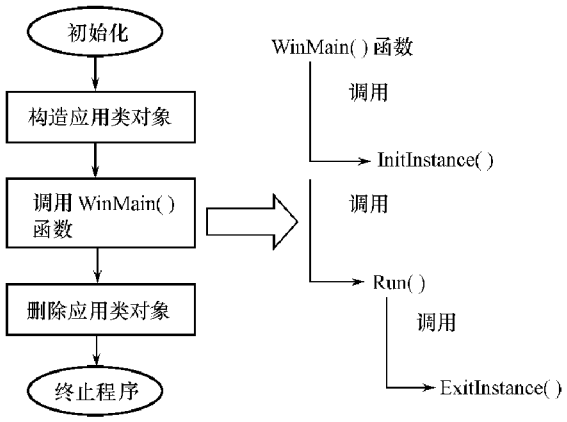



图 1.2　应用程序的运行过程

其中调用 WinMain( )函数又进一步细分为：

① 获得指向惟一应用类对象的指针。

② 进行一些内部初始化，若失败，则转至第(6)步。

③ 进行应用程序初始化，也就是调用应用类的 InitApplication( )函数进行初始化。如果失败则转至第(6)步，若成功则继续下一步。

④ 进一步初始化应用程序，即调用应用类 InitInstance( )函数进行初始化。由于 MFC AppWizard 自动生成的应用程序框架中的应用类重载了 MFC 中应用类基类的 InitInstance( )函数，所以这时调用的就是应用程序框架中应用类对象的 InitInstance( )函数。如果初始化失败，则调用应用类的 ExitInstance( )函数，然后转至第(6)步，若初始化成功则继续下一步。

⑤ 调用应用类的 Run( )函数，其主要功能就是运行消息循环，不断获取消息和处理消息（翻译和分派），若有用户行为如鼠标移动等消息，则处理这些消息；若没有，则进行空闲消息处理。一直到用户关闭应用程序窗口，产生 WM _ QUIT 消息时，Run( )函数才调用类的 ExitInstance( )函数，准备退出。

（5）终止应用程序。

（6）进行退出应用程序前的收尾工作，如删除注册的窗口类并释放其内存等。

（7）返回。

## 1.3　第一个 MFC 应用程序

用 Visual C++ 6.0 的 MFC 编写 Windows 应用程序，是一种“填空式”的编程方法，一般有 3 个步骤：

（1）创建工程　用 Visual C++ 6.0 的 MFC AppWizard 生成应用程序的工程文件，也就是创建应用程序的基本框架。

（2）可视化设计　用 Visual C++ 自带的工具软件 Winzards，制作 Windows 风格的图形用户界

面和各种控件，如矩形按钮、滚动条和单选按钮等。

（3）编写程序代码 根据目标程序的要求，用 MFC ClassWizard 添加消息响应函数，然后用 Visual C++ 提供的文本编辑器和 C++ 程序设计语言在函数中编写代码。

### 1.3.1 MyHello 应用程序

在开始编写 MyHello.exe 程序之前，首先观察一下它的外观和所具有的功能：

（1）例程 MyHello.exe 运行结果如图 1.3 所示。

（2）程序主窗口显示字符串：“Hello，我们开始 V C++ 编程了！”。

图 1.3 MyHello 程序运行结果

（3）单击 MyHello.exe 程序主窗口右上角关闭按钮，退出 MyHello.exe 程序。

对 MyHello 程序进行目标分解，用 Visual C++ 6.0 开发，由以下两步完成：

① 用 Visual C++ 6.0 的 MFC AppWizard，创建应用程序的基本窗口框架。

② 编写显示字符串：“Hello，我们开始 V C++ 编程了！”的代码。

### 1.3.2 创建工程

在生成 MyHello.exe 程序的工程文件和其他文件之前，首先应在硬盘上创建一个新目录，比如 d:\MYVC，以便存放程序所生存的文件。

创建 MyHello 工程的步骤如下：

（1）启动 Visual C++ 6.0。从“File”菜单中选择“New”。此时，Visual C++ 将显示一个“New”对话框。

（2）在“New”对话框中选择“Project”标签，然后选择“MFC AppWizard(exe)”类型，告诉 Visual C++ 即将创建一个 EXE 程序。

（3）在“New”对话框的“Project name”文本编辑框中输入“MyHello”，单击位于“Location”框右边的小按钮，再从下拉的对话框中选择“d:\MYVC”目录，使新创建的工程文件放置在“d:\MYVC”目录之下。

以上几个步骤分别指定了 MyHello.exe 程序的工程类型、工程名字和工程位置，此时“New”对话框如图 1.4 所示。

（4）单击“OK”按钮。此时 Visual C++ 将显示如图 1.5 所示的“MFC AppWizard - Step 1”对话框。

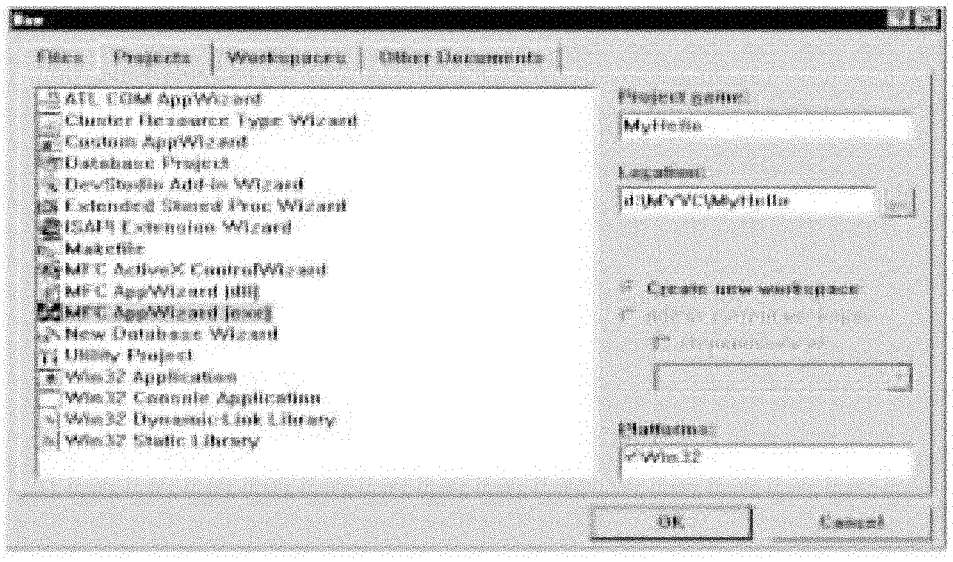



图 1.4　指定工程类型、工程名字和工程位置

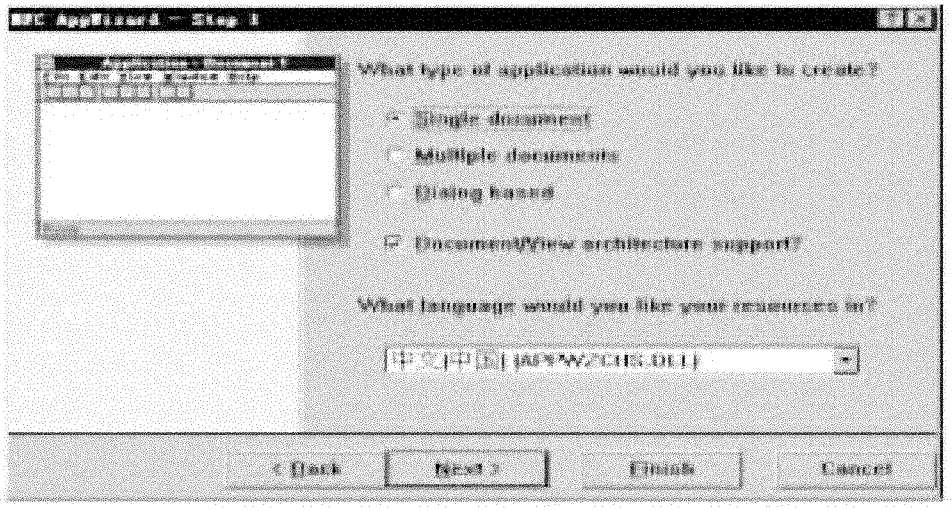



图 1.5　MFC AppWizard – Step 1：设置应用程序类型

（5）在“MFC AppWizard – Step 1”对话框中：

①“Single document”选项表示单文档界面，简称 SDI，这种类型应用程序的主窗口只能容纳一个文档，如 Windows 自带的记事本。

②“Multiple documents”选项表示多文档界面，简称 MDI，这种类型应用程序容许同时打开多个文档，这些文档可以层叠于主窗口。Microsoft Office 产品就属于 MDI 应用程序。

③“Dialog based”选项表示生成基于对话框的应用程序。

在本例中选择“Single document”，创建一个基于单文档界面的应用程序。然后选择资源语言，本例中选择“简体中文”。

（6）单击“Next”按钮，Visual C++ 6.0 显示“MFC AppWizard – Step 2 of 6”对话框，如图 1.6 所示。选择应用程将采用的数据库端口，因本例不需要数据库支持，接受默认的“None”选项。

（7）单击“Next”按钮，Visual C++ 6.0 显示“MFC AppWizard – Step 3 of 6”对话框，如图 1.7 所

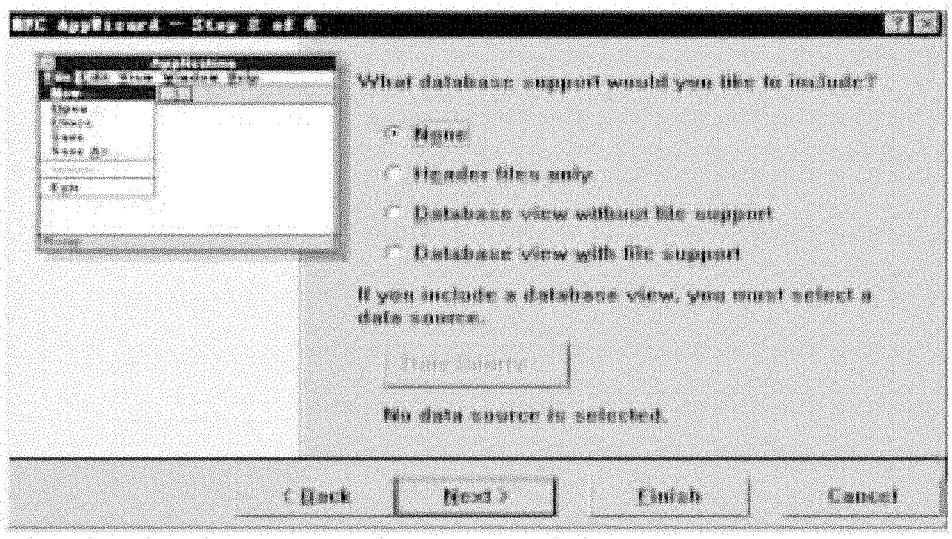



图 1.6 “MFC AppWizard – Step 2 of 6”对话框，设置数据库支持模式

示。其中是对应用程序的综合文档端口以及它端口的定义，接受默认设置。

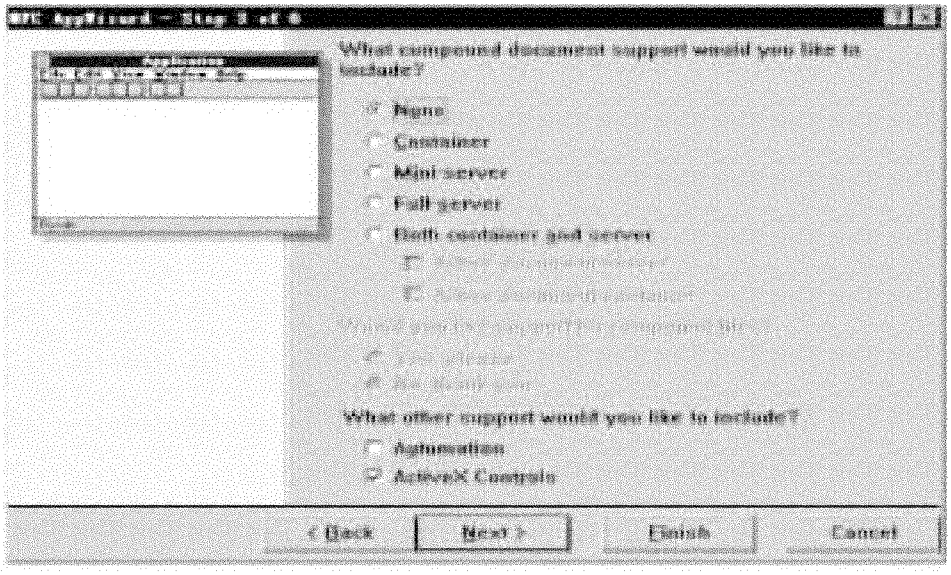



图 1.7 “MFC AppWizard-Step 3 of 6”对话框，设置 COM 模式

（8）单击“Next”按钮，Visual C++ 6.0 显示“MFC AppWizard – Step 4 of 6”对话框，如图 1.8 所示，接受默认设置。

（9）单击“Next”按钮，Visual C++ 6.0 显示“MFC AppWizard – Step 5 of 6”对话框，如图 1.9 所示。对话框中最下面的单选框有两个选项：“As a shared DLL”和“As a statically linked library”，前者表示将 MFC 库作为共享 DLL，这样生存的 MFC 可执行程序将小得多；后者表示静态链接 MFC，这样生存的程序会很大。对话框中各项接受默认设置。

（10）单击“Next”按钮，Visual C++ 6.0 显示“MFC AppWizard – Step 6 of 6”对话框，如图 1.10 所示。其中显示 MFC AppWizard 为应用程序创建的所有类以及各个类对应的基类和相应的文件，接受默认设置。

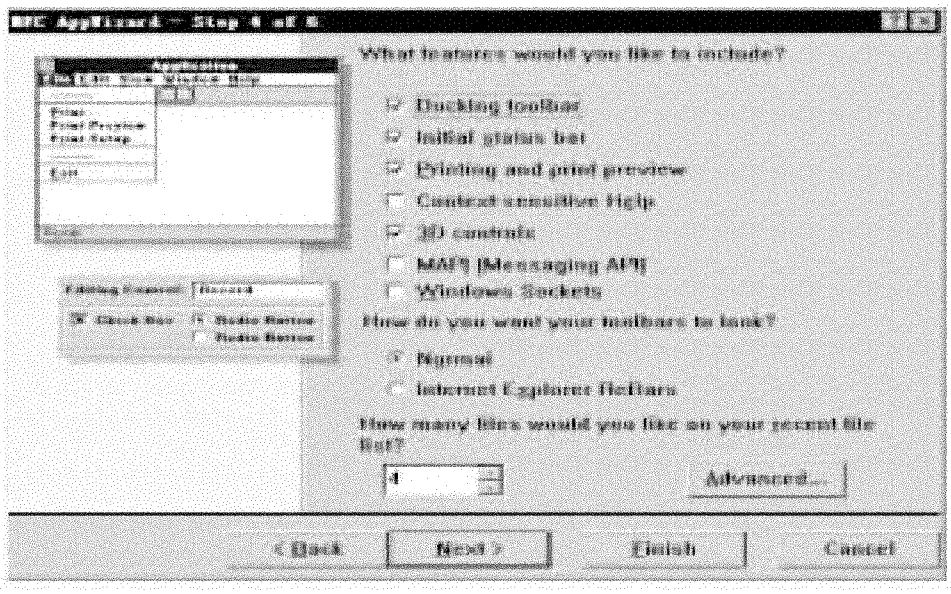



图 1.8　“MFC AppWizard – Step 4 of 6”对话框：设置属性选项

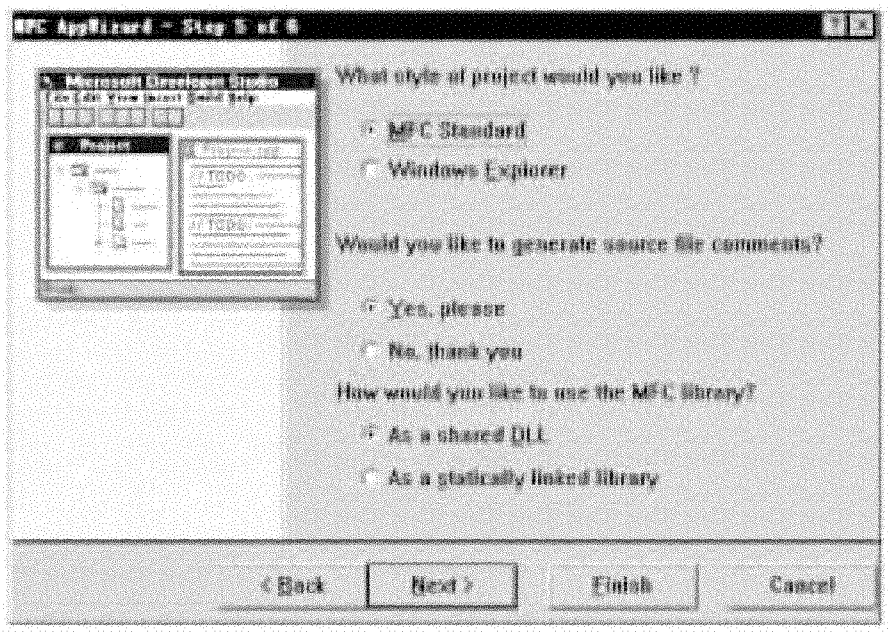



图 1.9　“MFC AppWizard – Step 5 of 6”对话框：设置项目样式、注释和 MFC 库

特别说明：

① 本例是为读者了解 MFC AppWizard 各步的作用，所以逐步详述。如果使用后几步骤的默认设置，则可单击“Finish”按钮，结束设计工作。

② 如想修改前面某步的设置，请单击“Back”按钮，可重新设置。

（11）单击“Finish”按钮，结束 MFC AppWizard 的设计工作，此时 Visual C++ 将显示“New Project Information”窗口，如图 1.11 所示。窗口中显示前 6 步所做的选择的汇总信息。

（12）单击“OK”。于是，Visual C++ 就创建了 MyHello 工程以及相关的所有文件。

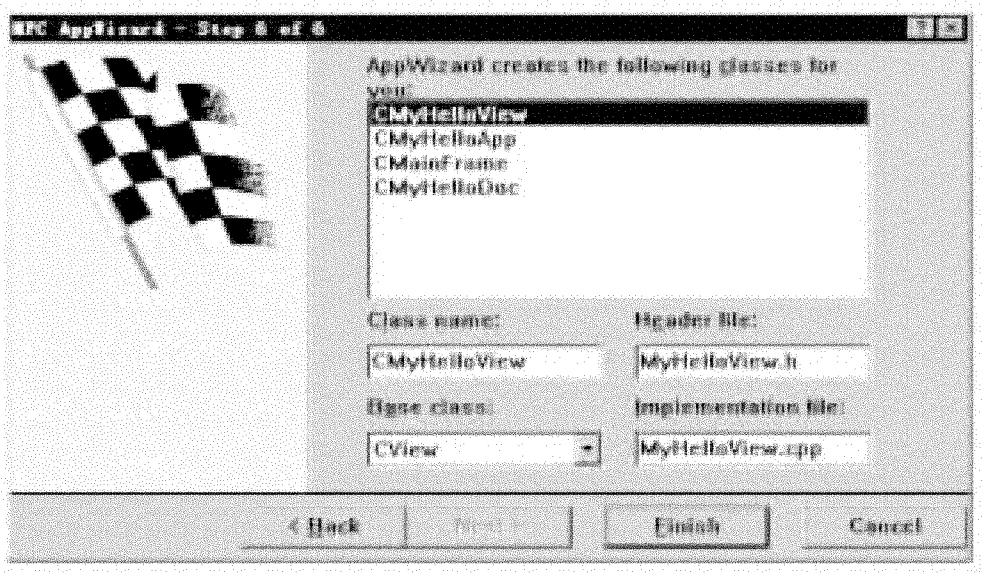



图 1.10 “MFC AppWizard – Step 6 of 6”对话框：设置基类

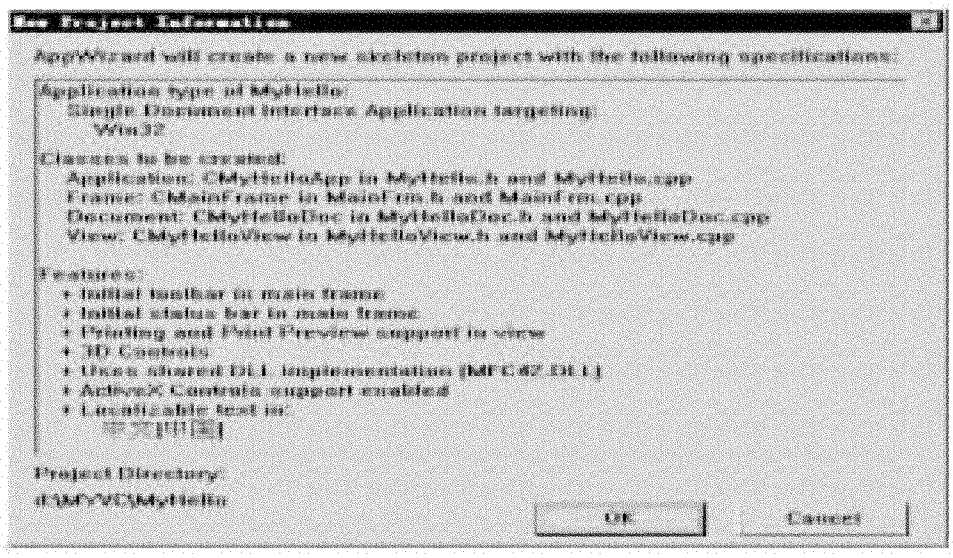



图 1.11 “New Project Information”对话框

### 1.3.3 编写程序代码

用 Visual C++ 6.0 编写 MFC 应用程序，是一种“填空式”的编程方法，因为利用 MFC AppWizard 生成框架程序，使得程序员免去了大量的模式化工作。在生成框架程序后，用户只要根据目标程序的要求，看哪些地方需要修改，再往里填写必要的代码。

接下来就要添加自己的代码了，在“Workspase”窗口的“FileView”标签（工程资源管理器）中选择 MyHelloView.cpp 文件，双击打开，然后找到函数 OnDraw(CDC * pDC)，在其中添加显示字符串：“Hello，我们开始 V C++ 编程了！”的代码。

```
void CMyHelloView::OnDraw(CDC* pDC)
```

```
{
    CMyHelloDoc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    // TODO: add draw code for native data here
    pDC->TextOut(100,80,"Hello，我们开始V C++ 编程了!");
}
```

这样的一个语句的功能就是用来在屏幕坐标的(100,80)处显示出字符串“Hello，我们开始V C++ 编程了!”。有一点要指出的是，在输入上面的那行代码的时候，除了双引号中的文字随意外，其余的一定要保证是英文方式下的字符。否则，程序编译时会出错，用户能对这样所导致的一大堆错误感到束手无策。

### 1.3.4　编译运行 MyHello 应用程序

代码加上之后，就可以对程序进行编译、连接和运行了，操作步骤如下：

（1）如图 1.12 所示，选择“Build”菜单中的“Build MyHello.exe”菜单项，Visual C++ 就会编译并连接成 MyHello.exe 程序。

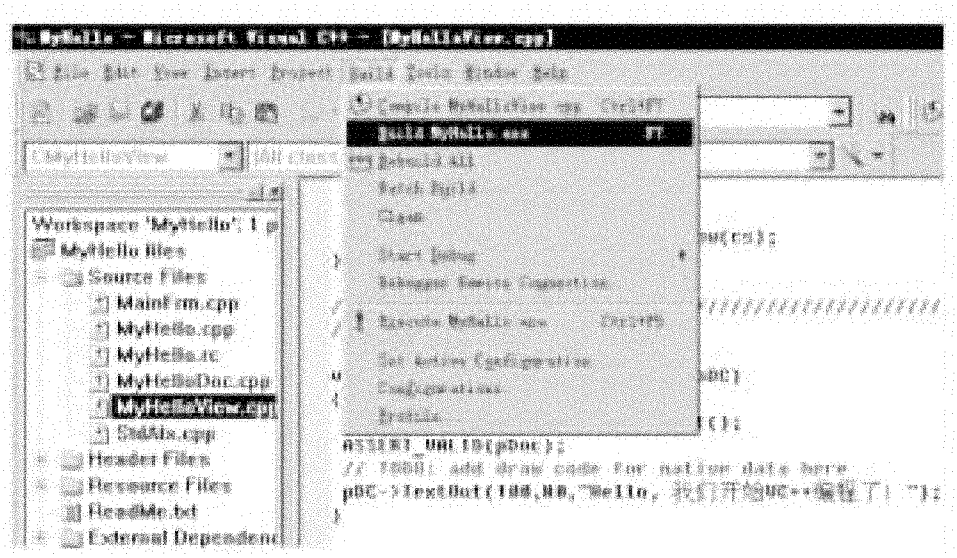


图 1.12　编译、连接

（2）选择“Build”菜单中的“Execute MyHello.exe”菜单项，Visual C++ 就会执行 MyHello.exe 程序，图 1.3 所示的 MyHello.exe 程序主窗口也随之出现。

（3）单击 MyHello.exe 程序主窗口右上角关闭按钮，退出 MyHello.exe 程序。

## 1.4　应用程序分析

从前面的叙述可以看到，用 MFC AppWizard 来帮助生成程序时，MFC 类做了很多工作，使用户很容易地就能够编写一个程序。下面来具体看一下 MFC 到底做了些什么，使大家对用 MFC 编程有一个大致的认识。

打开“Workspace”窗口中的“ClassView(类视图)”标签,可以看到 MFC 生成了 5 个类,如图 1.13 所示。

- CAboutDlg
- CMainFrame
- CMyHelloApp
- CMyHelloDoc
- CMyHelloView

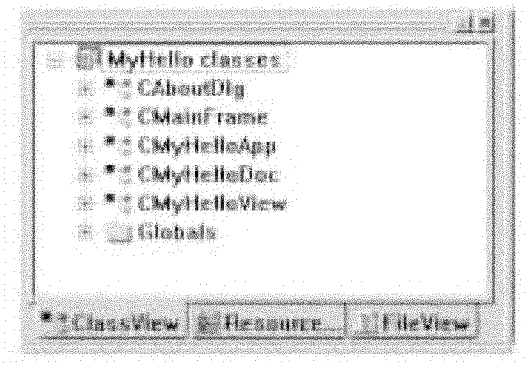



图 1.13 MyHello 工程的类视图

对应每一个类 MFC Wizard 生成了两个文件,分别以 .h 和 .cpp 结尾。在 .h 文件中 MFC Wizard 定义了类和它的各种变量、函数;在 .cpp 文件中来生成类对象,实现这个类。

下面依此来分析类和文件。首先声明一下,这些代码是 MFC Wizard 自动生成的,在看时只要大体了解它们的功能就可以了,现在不需要做深入细致的分析,更不要为现在看不懂它们而着急。还要声明一点,由于只是了解一下它们的大致结构功能,故这里只给出类及函数的声明部分,而对 .cpp 文件暂不做分析。

### 1.4.1 应用类 CMyHelloApp

应用类 CMyHelloApp 派生于 MFC 中的 CWinApp 类,可以在 MyHello.h 文件中看到这一点,这个类的作用是初始化应用程序及运行该应用程序所需要的成员函数。而 CWinApp 类派生于 CWinThread 类,代表了程序中运行的主线程,它就是运行程序的本身,所以每一个基于 MFC 创建的应用程序只能包含该类惟一的派生类对象。

MyHello.h 是应用程序的主头文件,它声明了 CMyHelloApp 类,需要了解的是虚函数 InitInstance()的作用。该函数用于设置注册数据库,载入标准设置(最近打开文件列表等)、注册文档模板,其中在注册文档模板过程中隐含地创建了主窗口;接下来,该函数处理命令行参数,显示窗口,然后返回、进入消息循环。MyHello.h 的完整代码如下:

```
// MyHello.h : main header file for the MYHELLO application
#pragma once
#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif
#include"resource.h"       // main symbols

// CMyHelloApp:
// See MyHello.cpp for the implementation of this class
class CMyHelloApp : public CWinApp
{
public:
    CMyHelloApp();
// Overrides
    // ClassWizard generated virtual function overrides
```

```
    //{{AFX_VIRTUAL(CMyHelloApp)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation
    //{{AFX_MSG(CMyHelloApp)
    afx_msg void OnAppAbout();
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

### 1.4.2　主框架窗口类 CMainFrame

窗口类 CMainFrame 派生于 CFrameWnd，它主要用来管理应用程序的窗口，显示标题栏、工具栏、状态栏等。同时它还要处理针对窗口操作的消息。视图窗口是主框架窗口的一个子集，对于多文档来说，主框架窗口是所有多文档应用子窗口的容器。

查看 MainFrm.h 代码，能够看到在类 CMainFrame 中声明的两个主要函数 PreCreateWindow() 和 OnCreate()以及两个对象 m_wndStatusBar 和 m_wndToolBar。

其中对象 m_wndStatusBar 属于状态栏类 CStatusBar，它用于创建管理状态栏；m_wndToolBar 属于工具栏类 CToolBar，它用于创建管理工具栏。MainFrm.h 的完整代码如下：

```
// MainFrm.h : interface of the CMainFrame class
#pragma once
class CMainFrame : public CFrameWnd
{

protected: // create from serialization only
    CMainFrame();
    DECLARE_DYNCREATE(CMainFrame)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
```

```
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;

// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

### 1.4.3 文档类 CMyHelloDoc

文档类 CMyHelloDoc 派生于 CDocument 类，该类主要用来存放应用程序的数据以及文件的保存加载功能。文档类要通过视图类来实现与用户的交互。

CMyHelloDoc.h 中声明的 OnNewDocument()函数用于初始化文档，Serialize()函数串行化(保存和装入)文档，Dump()函数用于调试诊断。详细代码如下：

```
// MyHelloDoc.h : interface of the CMyHelloDoc class
#pragma once
class CMyHelloDoc : public CDocument
{
protected: // create from serialization only
    CMyHelloDoc();
    DECLARE_DYNCREATE(CMyHelloDoc)

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyHelloDoc)
    public:
    virtual BOOL OnNewDocument();
    virtual void Serialize(CArchive& ar);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMyHelloDoc();
#ifdef _DEBUG
    virtual void AssertValid() const;
```

```
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CMyHelloDoc)
        // NOTE - the ClassWizard will add and remove member functions here.
        // DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

### 1.4.4　视图类 CMyHelloView

视图类 CMyHelloView 派生于 CView 类，视图类用于管理视图窗口，它对应的对象在框架窗口中实现用户数据的显示和打印。

MyHelloView.h 文件中声明了与文档数据相关的 3 个函数 OnPreparePrinting()、OnBeginPrintting()和 OnEndPrinting()，用以实现数据打印；声明了返回 CMYhelloDoc 指针的函数 GetDocument()，用以获取文档的指针，以实现对用户文档的数据的操作；声明了函数 OnDraw()，用以实现视图数据的显示和刷新。详细代码如下：

```
// MyHelloView.h : interface of the CMyHelloView class
#pragma once
class CMyHelloView : public CView
{
protected: // create from serialization only
    CMyHelloView();
    DECLARE_DYNCREATE(CMyHelloView)

// Attributes
public:
    CMyHelloDoc* GetDocument();


// Operations
public:


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMyHelloView)
    public:
    virtual void OnDraw(CDC* pDC); // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
```

```
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMyHelloView();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CMyHelloView)
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

### 1.4.5 预编译头文件 stdafx.h

stdafx.h 用于建立一个预编译的头文件 MyHello.pch 和一个预定义的类型文件 stdafx.obj，由于 MFC 体系结构非常大，如果每次都编译的话比较费事，因此，把常用的 MFC 头文件都放在 stdafx.h 中，如 afxwin.h、afxext.h、afxdisp.h 等，然后让 stdafx.cpp 包含 stdafx.h 文件。这样由于编译器可以识别哪些文件已经编译过，所以 stdafx.cpp 就只编译一次。因为它存放的是头文件编译后的信息，故称做预编译头文件。如果读者以后在编程时不想让有些 MFC 头文件每次都被编译，也可以将它加入到 stdafx.h 中，采用预编译头文件，可以加速编译过程。它的源代码如下：

```
// stdafx.h : include file for standard system include files,
// or project specific include files that are used frequently, but
//are changed infrequently
#if !defined(AFX_STDAFX_H__F67329A7_518D_11D7_829C_C8CB72F94C47__
INCLUDED_)
#define AFX_STDAFX_H__F67329A7_518D_11D7_829C_C8CB72F94C47__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN    // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>     // MFC core and standard components
#include <afxext.h>     // MFC extensions
#include <afxdisp.h>    // MFC Automation classes
#include <afxdtctl.h>   // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>     // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual  C++ will insert additional declarations immediately before
the previous line.

#endif
// !defined(AFX_STDAFX_H__F67329A7_518D_11D7_829C_C8CB72F94C47__INCLUDED_)
```

### 1.4.6　资源文件

除了上述几个重要部分外，MFC AppWizard 还会创建一些与资源相关的文件，这些都包含在资源文件中，资源文件含有一般 MFC 应用程序的默认菜单定义和加速键表、字符串表。它还指定了默认的 About 对话框和一个图标文件(RES \MyHello.ico)。通常的资源文件有：

(1) MyHello.ioc　此图标文件是应用程序的图标，它包括在主资源文件 MyHello.rc 中。

(2) MyHello.rc2　此文件存放 Visual Studio 不可直接编辑的资源，用户应该将任何不能直接被 Visual Studio 编辑的文件放到此文件中。

(3) Toolbar.bmp　这个文件中包含了工具栏所用到的位图资源。可以用资源编辑器来编辑这些位图，要使用它们还要在工具栏数组中添加它们，同时要添加工具栏按钮。

(4) MyHelloDoc.ico　该文件是生成的文档的图标。

以下是资源文件的头文件 Resource.h，它里面给出了有关各种资源 ID 和键值的描述。

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual  C++ generated include file.
// Used by MYHELLO.RC
//
#define IDD_ABOUTBOX                    100
#define IDR_MAINFRAME                   128
#define IDR_MYHELLTYPE                  29
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_3D_CONTROLS1
#define _APS_NEXT_RESOURCE_VALUE                130
#define _APS_NEXT_CONTROL_VALUE                 1000
#define _APS_NEXT_SYMED_VALUE                   101
#define _APS_NEXT_COMMAND_VALUE                 32771
#endif
#endif
```

(5) ReadMe.txt 文件　此文件包含的是关于该应用程序的说明，包括各个文件的功能介绍，事实上上述的各文件的介绍也就是此文件的转义而已。

# 习 题 一

1. Windows 程序设计的特点是什么？

2. Windows 应用程序的优点是什么？

3. MFC AppWizard(应用向导)可以创建哪几类应用程序？各有何特点？

4. 请用 MFC AppWizard 创建一个基于对话框的应用程序，列出所创建的类，并与 MyHello 工程比较，观察它们的异同。

# 第2章

# MFC程序的界面设计与资源管理

本章导读

应用程序是为用户开发的，所以用户界面必须友善、易于操作。如何设计简洁明了的界面，使用户能够轻松自如地工作，是程序设计中要解决的一个重要问题。

应用程序的界面设计中涉及到诸如菜单、对话框、消息框以及按钮等标准格式数据，Windows将这些数据保存在资源文件中，将它们作为资源来进行管理，所以，MFC程序的界面设计就是对资源的创建和修改编辑。

通过本章的学习，使读者掌握：

- MFC程序的界面设计　包括菜单、工具栏上按钮、对话框等
- 资源的创建与维护　包括鼠标资源、快捷键、图标、菜单、工具栏等

---

## 2.1　资源与界面

程序员设计任何应用程序均将涉及到诸如菜单、对话框、消息框以及按钮等标准格式数据。Windows将这些数据保存在资源文件中，程序员可通过编辑工具编辑、修改这些资源文件，使其提供所需的菜单或按钮，并将其放入设计的程序之中。Windows 95/98/2000将应用程序常用到的一些“数据”归纳定义成可共享的资源，例如Visual C++ 6.0中将某些静态的、可归类的和可共享的数据以资源的形式处理，实现了可见即可得的可视化目标。

Visual C++ 6.0可以处理的资源有：菜单(Menu)、图标(Icon)、对话框(Dialog)、位图(Bitmap)、快捷键(Accelerator)、工具栏(Toolbar)、光标(Cursor)、版本描述(Version)和字符串表(String Table)。

以资源作为线索，增强了编程的逻辑性，简化了开发过程。其中还为编辑菜单、对话框、工具栏等资源的程序界面设计提供了直观、可视的设计方法，即程序的界面设计在很大程度上就是对相关资源的编辑处理。因而熟练掌握资源编辑器，对界面设计及了解整个应用程序资源的逻辑关系都是非常重要的。

# 2.2 资源管理

资源管理包括各种资源的创建与维护，Visual C++6.0 中提供了一整套管理资源的方法，通过 Workspace 的“Resource View”(资源视图)的向导，可以创建资源，或激活各种资源相应的资源编辑器，以进行可视化编辑。

## 2.2.1 应用程序的打开与关闭

Visual C++6.0 对应用程序的管理是以工程形式组织的，而一个工程是由多个文件组成，所以，打开保存在磁盘上的应用程序，就是打开应用程序工程，常用的方法有如下两种。

**1.** 历史工程选择法

这是最简捷、最快速的方法，操作步骤如下：

(1) 选择“File”菜单下的“Recent Workspace”菜单项，Visual C++6.0 将列出最后打开过的历史工程，如图 2.1 所示。

(2) 在历史工程列表中，选择要打开的应用程序工程，比如 MyHello，Visual C++6.0 就将该工程调入开发环境。

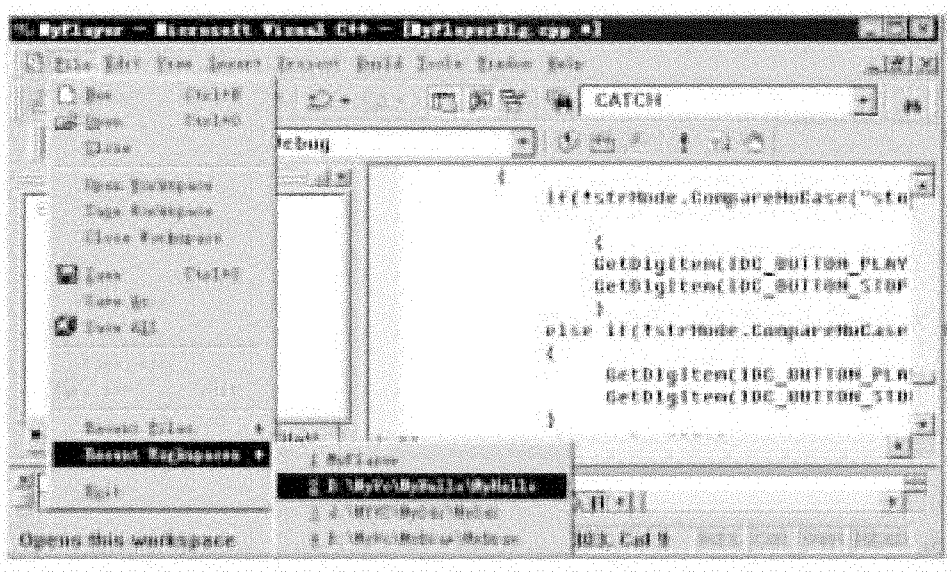



图 2.1 历史工程选择法

**2.** 直接选择法

这是最原始的常用方法，操作步骤如下：

(1) 选择“File”菜单下的“Open Workspace”菜单项，Visual C++6.0 将弹出“Open Workspace”对话框。

(2) 在“Open Workspace”对话框中，查找所需的应用程序，选中 *.dsw(比如，选中 MyHello.dsw)文件，单击“打开”按钮，如图 2.2 所示，就将所选工程调入 Visual C++6.0 开发环境。

特别说明：

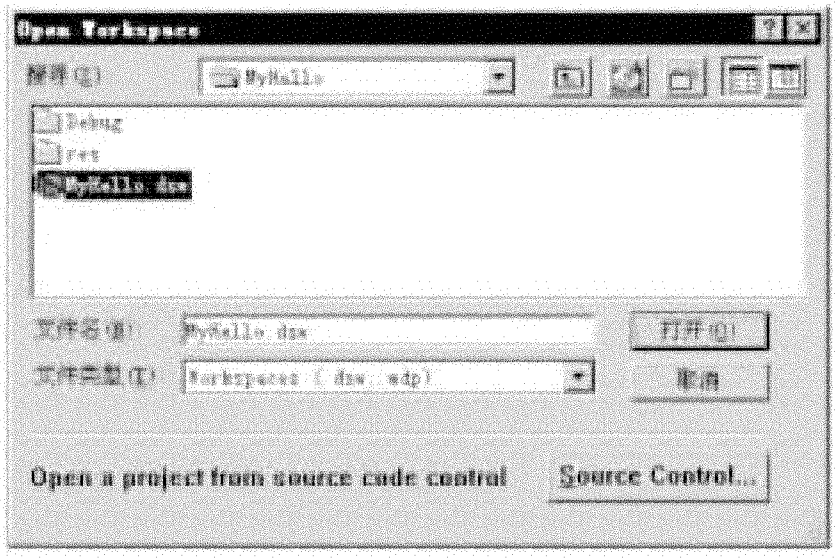



图 2.2　“Open Workspace”对话框

因为 Workspace 是应用程序的类、资源、文件等全部信息的“浏览器”，因此，在打开 Workspace 的同时，Visual C++ 6.0 将相应的应用程序调入 Visual C++ 6.0 开发环境中。

而关闭应用程序，操作极为简单，有以下两种情况：

一是 选择“File”下拉菜单中的“Close Workspace”选项，按向导提示进行操作，就可关闭已打开的应用程序工程。另一种情况是再次打开某应用程序工程或创建新的应用程序工程后，Visual C++ 6.0 都会自动将当前的应用程序工程放入历史工程的列表中。

### 2.2.2　浏览应用程序资源

应用程序打开后，其资源由“Workspace”窗口来组织管理、浏览，在“Workspace”窗口中包含“ClassView”、“ResourceView”和“FileView”3 个视图标签，如图 2.3 所示。“ResourceView”视图标签

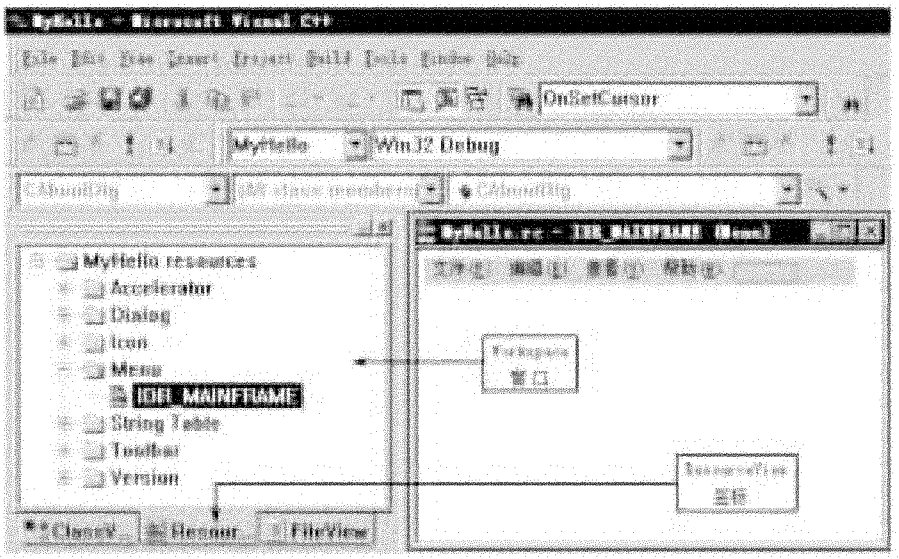



图 2.3　Workspace 窗口

管理应用程序所涉及的资源。如果“Workspace”窗口未打开，则在“View”下拉菜单中选择“Workspace”选项。

在本例(MyHello 工程)中包含的资源类有：Accelerator(快捷键)、Dialog(对话框)、Icon(图标)、Menu(菜单)、String Table(字符串表)、Toolbar(工具栏)和 Version(版本描述)。各资源类中包含有具体资源项，这些资源项组成了应用程序自己的资源，程序员也可添加(或删除)资源或资源项。

### 2.2.3 增加新资源

为工程增加不同类型的资源的方法基本相同。例如，为 MyHello 工程增加对话框资源，其操作步骤是：

(1) 右击 Workspace 中资源项，Visual C++6.0 显示如图 2.4 所示的快捷菜单。

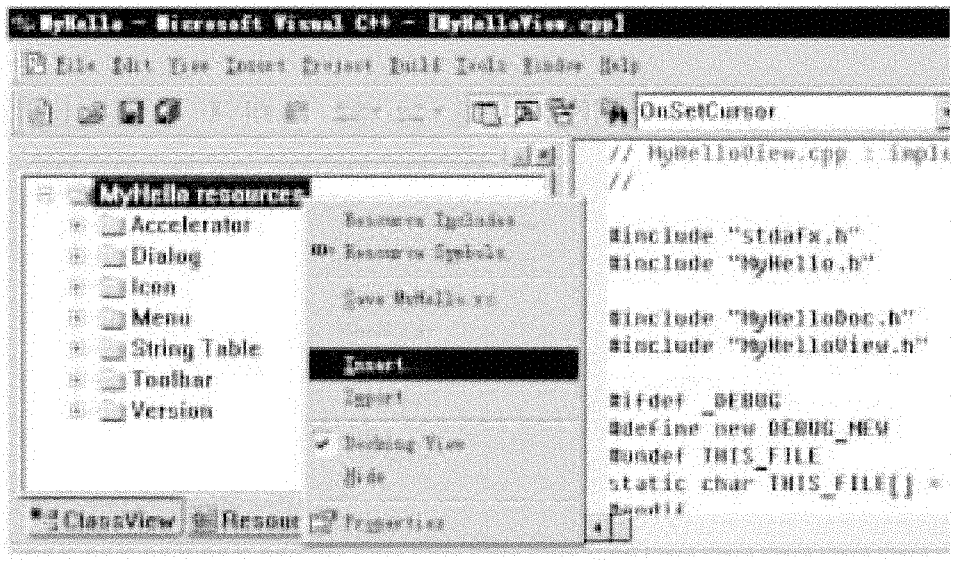



图 2.4 快捷菜单

(2) 选择“Insert”选项，Visual C++6.0 显示“Insert Resource”对话框，选择所需资源，例如，选择“Dialog”(对话框)资源，如图 2.5 所示。

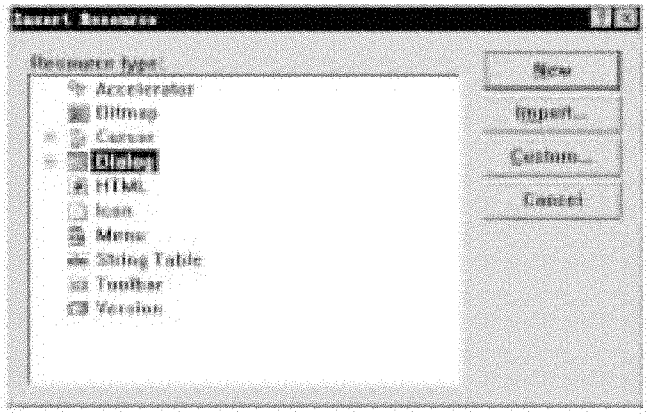



图 2.5 “Insert Resource”对话框

（3）单击“New按钮”，就在资源列表中新增加了“Dialog”资源项，也就是新建一个ID（标记符）为“IDD_DIALOG1”的对话框资源项，如图2.6所示。

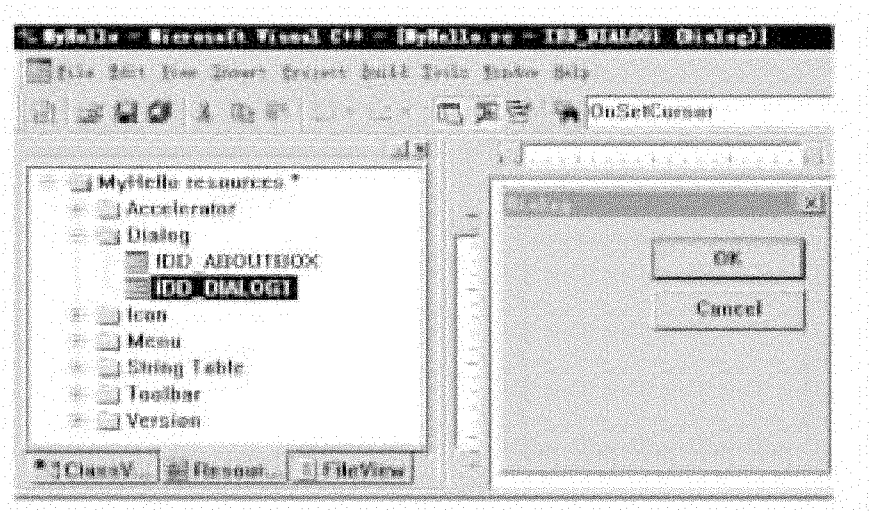



图2.6　新增加的“Dialog”资源

完全类似，可以为工程增加其他类型资源，比如，菜单、图标等资源。作为练习，请参考本章习题。

### 2.2.4　删除资源

删除资源就是将某一资源从工程中删除，操作方法是：在“Workspace”窗口的“ResourceView”标签中，用鼠标选中要删除的资源ID（标识符），按键盘上的“Del”键即可。

例如，在图2.6中，选中“Dialog”下的“IDD_DIALOG1”标识，按下“Del”键，标识为IDD_DIALOG1的资源就被删除。

## 2.3　资源编辑器

MFC程序的界面设计就是对资源的创建、修改和编辑。Visual C++6.0提供了功能强大且易于使用的资源编辑器。无论是创建应用程序工程伴随的资源，还是新添加的资源，都与用户界面所需要的资源相差甚远。所以，需要编辑处理Visual C++6.0提供的资源，才能得到应用程序所需的资源。

本节将简要介绍如何使用Visual C++6.0提供的资源编辑器，对资源进行编辑。

### 2.3.1　快捷键（Accelerator）

快捷键是提高应用程序效率的常用方法，快捷键资源项的功能就是定义应用程序中的事件或对象与键盘键的对应关系。

例如，给工程MyHello添加快捷键的操作步骤如下：

（1）在Workspace窗口的“Resource View”标签中，双击“Accelerator”下的“IDR_MAINFRAME”

项，Visual C++6.0 显示如图 2.7 所示的快捷键资源列表。

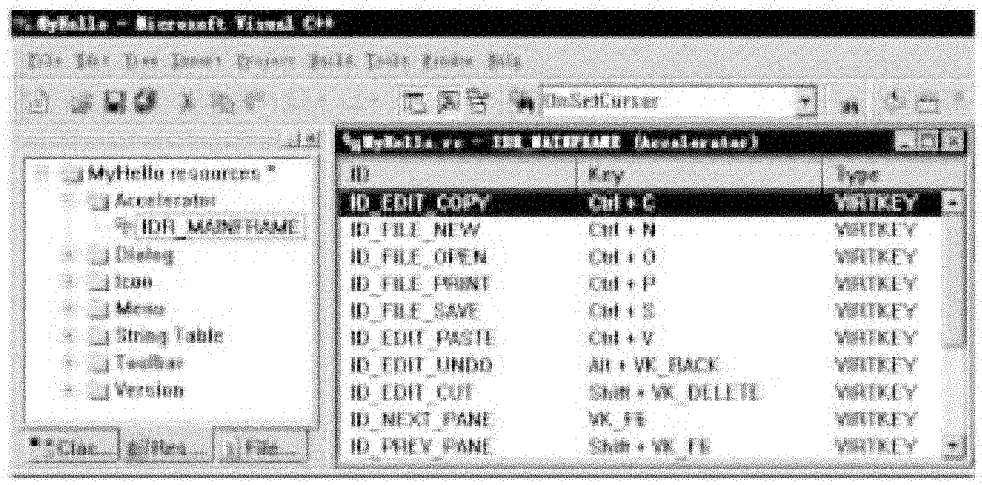



图 2.7 快捷键资源列表

（2）在快捷键资源列表窗口，右击鼠标弹出快捷菜单，如图 2.8 所示。

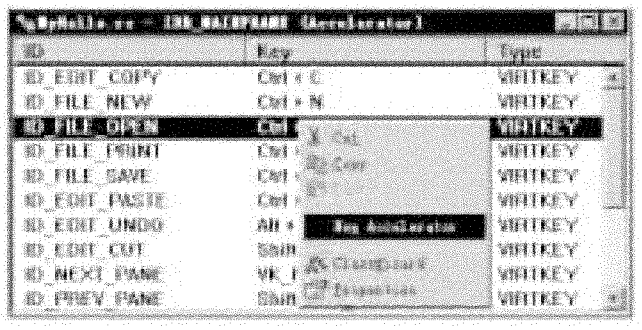


图 2.8 快捷菜单

（3）选择“New Accelerator”，Visual C++6.0 显示一个“Accel Properties”对话框，如图 2.9 所示，将增加一条快捷键对应关系。或者选择“Properties”来修改、编辑当前的快捷键的对应关系。

（4）在“Accel Properties”（快捷键属性）对话框中，定义了快捷键的对应关系。在本例中将 ID_EDIT_CUT 对象与“Ctrl + X”建立对应关系，这样在应用程序运行时，按“Ctrl + X”组合键即可执行 ID_EDIT_CUT 项。具体操作是：在“Key”下拉列表中选择“X”键或直接输入“X”，选择“Ctrl”复选框。

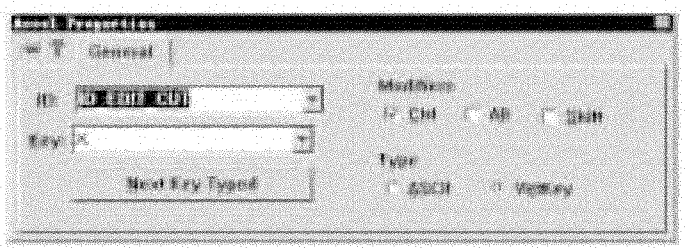


图 2.9 “Accel Properties”对话框

### 2.3.2　对话框(Dialog)

对话框设计是界面设计中最重要的部分,在MFC应用程序中,对话框资源的操作包括新建和编辑两种情况。

**1.** 创建对话框资源

在MyHello工程中,新建一个ID(标识符)为"IDD _ MYDIALOG"、"Caption"为"我的对话框"的对话框资源,其操作步骤如下:

(1) 在"Workspace"窗口中,激活"Resource View"标签,选中"Dialog"资源,右击鼠标,Visual C++6.0显示快捷菜单,如图2.10所示。

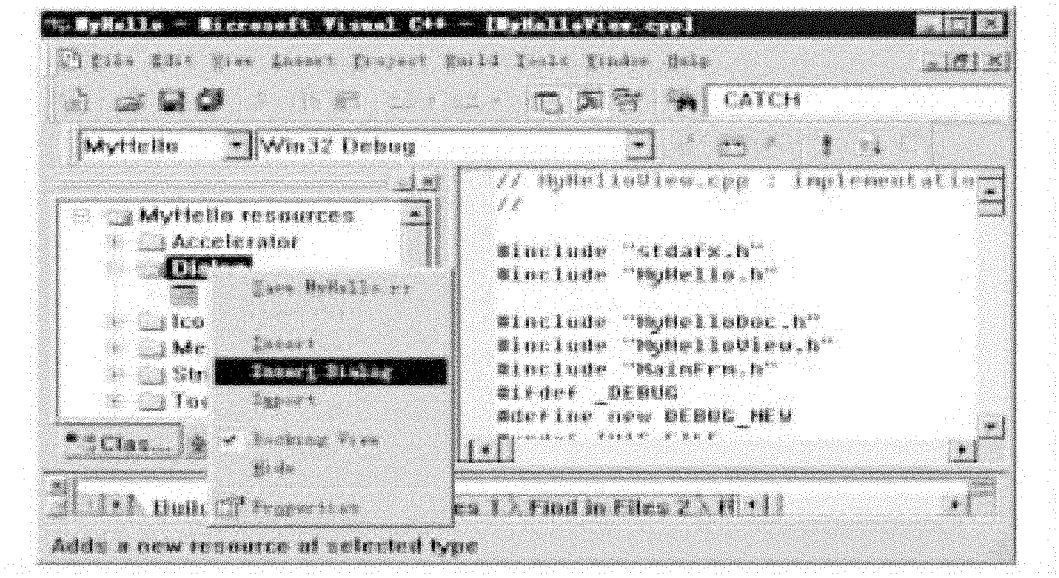



图2.10　快捷菜单

(2) 选择"Insert Dialog"选项,增加新的对话框。对话框的标识自动设为IDD _ DIALOG1,并同时弹出绘图工具箱窗口,如图2.11所示。

(3) 选中右侧新建的对话框资源,右击鼠标弹出快捷菜单,如图2.12所示。

(4) 选择"Properties"菜单项,弹出属性对话框,在"ID"文本编辑框中,将IDD _ DIALOG1修改为IDD _ MYDIALOG,"Caption"文本编辑框中的Dialog修改为"我的对话框",如图2.13所示。

(5) 点击非属性对话框上的任意点,关闭属性对话框,就创建了所需要的对话框资源,如图2.14所示。

**2.** 编辑对话框资源

对话框资源创建后,就可以利用Visual C++提供的编辑工具,按用户界面要求,进行可视化设计了。

例如,在"IDD _ MYDIALOG"对话框中添加一个ID为"ID _ MYBUTTON"、"Caption"为"我的按钮"的Button控件,操作步骤如下:

(1) 在"Project Workspace"窗口中,单击"ResourceView"标签,把"MyHello resources"扩展开,然

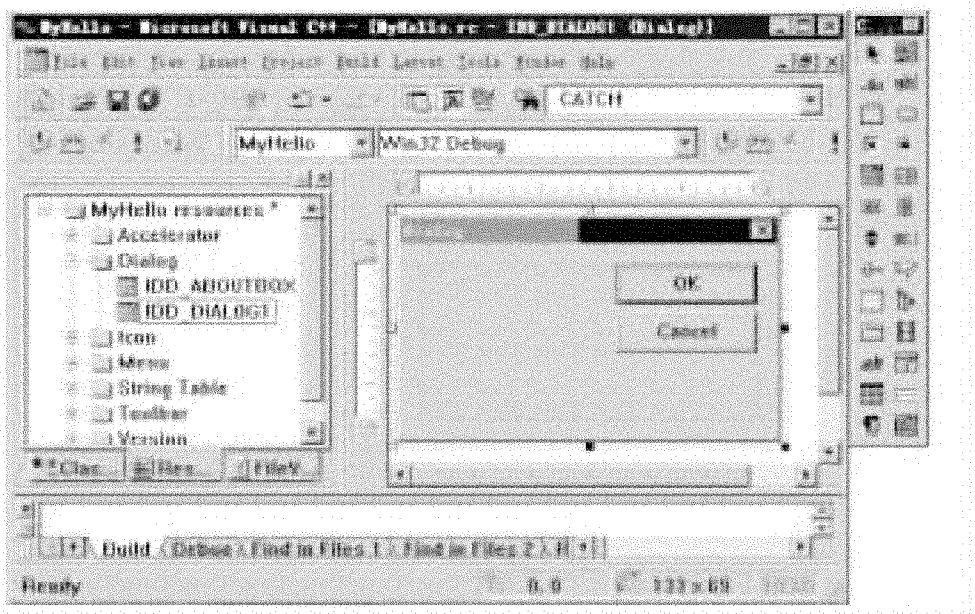



图 2.11 新建了一个“IDD _ DIALOG1”对话框资源

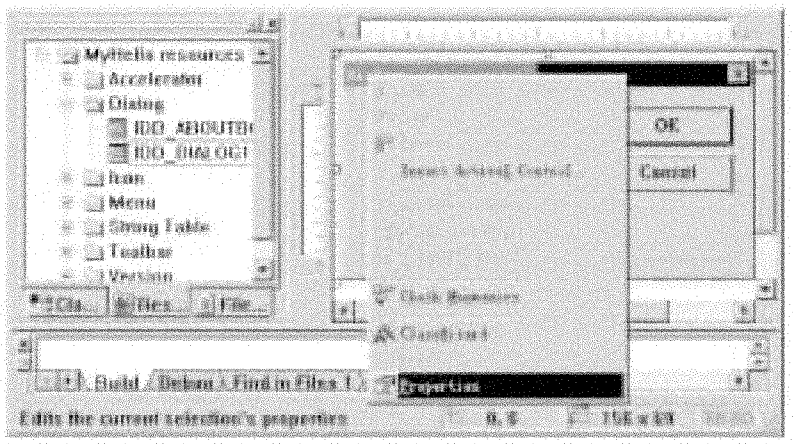



图 2.12 快捷菜单

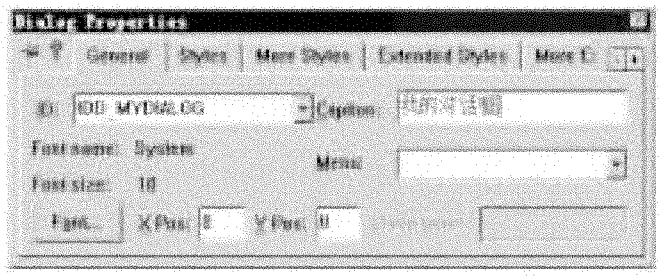



图 2.13 属性对话框

后再扩展 Dialog，最后，双击“IDD _ MYDIALOG”项，Visual C++ 显示出处于设计状态的“IDD _ MYDIALOG”对话框。

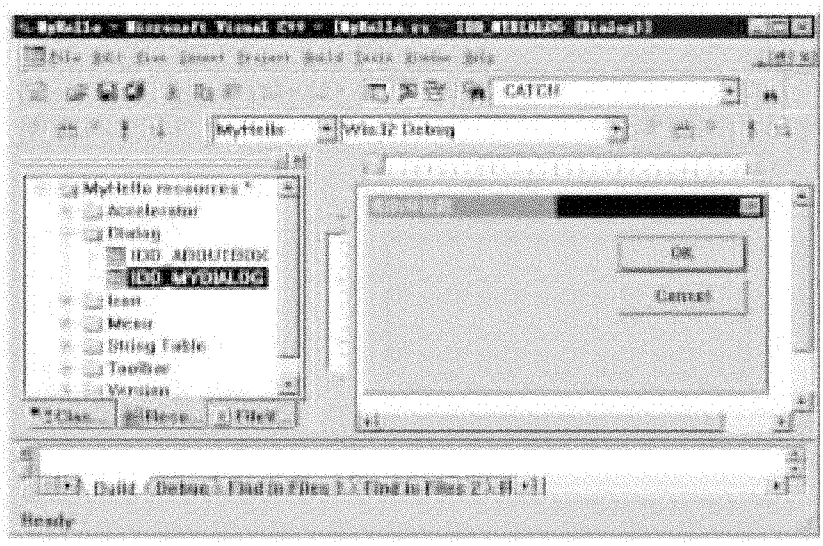



图2.14　创建好的对话框资源

（2）从“IDD _ MYDIALOG”对话框中删除“OK”和“Cancel”按钮。

（3）将鼠标放置工具栏的任一位置，单击鼠标右键弹出快捷菜单，如图2.15(a)所示。选择“Controls”复选项，将弹出矩形控件工具箱对话框，如图2.15(b)所示。

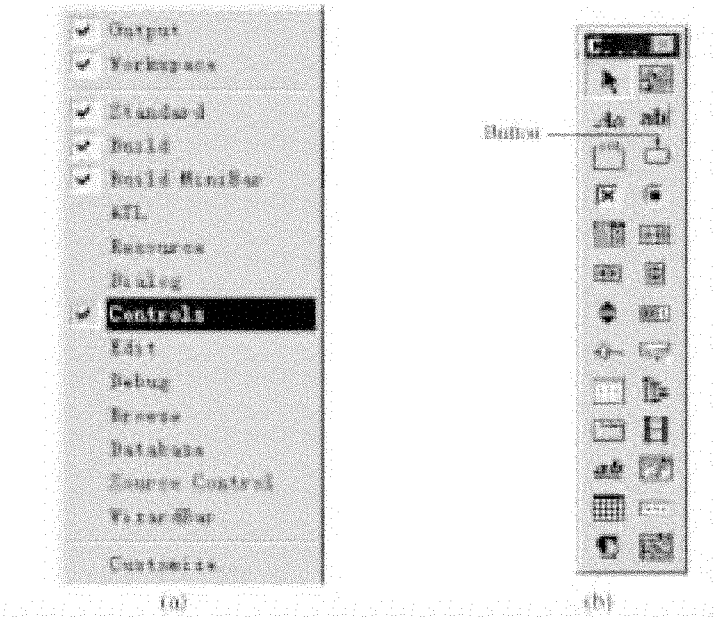



图2.15　工具栏和工具箱

（4）选中控件工具箱中的“Button”按钮控件，再单击“IDD _ MYDIALOG”对话框中适当的地方，Visual C++就会把“Button”按钮放置在刚才单击的地方，默认按钮标题是“Button1”，下面把其标题改为“我的按钮”。

（5）右击“Button1”按钮，将弹出如图2.16所示的快捷菜单。

（6）选择弹出菜单的“Properties”(属性)项，将显示“Push Button Properties”对话框，如图2.17所示。

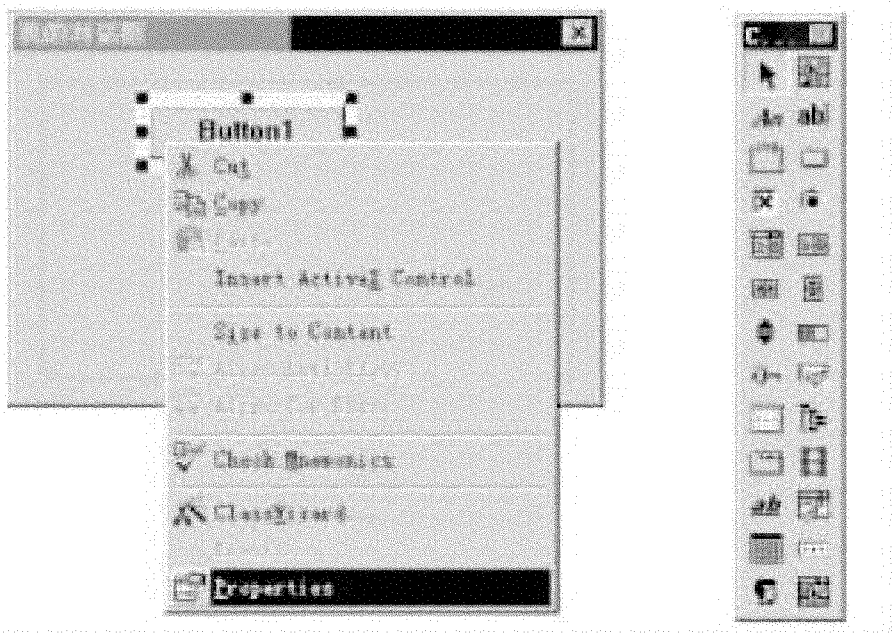



图 2.16 右击“Button1”按钮弹出的快捷菜单

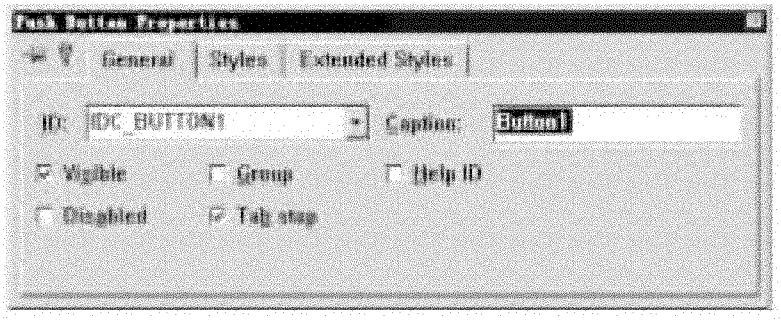



图 2.17 “Push Button Properties”对话框

（7）单击“ID”文本框，把文本“IDC _ BUTTON1”修改为“IDC _ MYBUTTON”，同样将“Caption”文本编辑框中的“Button1”修改为“我的按钮”。

至此，在“IDD _ MYDIALOG”对话框中添加了一个名为“我的按钮”的 Button 按钮如图 2.18 所示。

图 2.18 添加一个 Button 按钮

### 2.3.3 图标(Icon)

图标设计比较简单，类似位图的设计。实现方法是在 Workspace 的“Resource View”中右击“icon”资源项，在激活的菜单中选择“Insert icon”，插入新图标或双击存在的图标，打开如图 2.19 所示的图标资源编辑器，利用“Graphics”和“Colors”工具箱，编辑所要的图标。

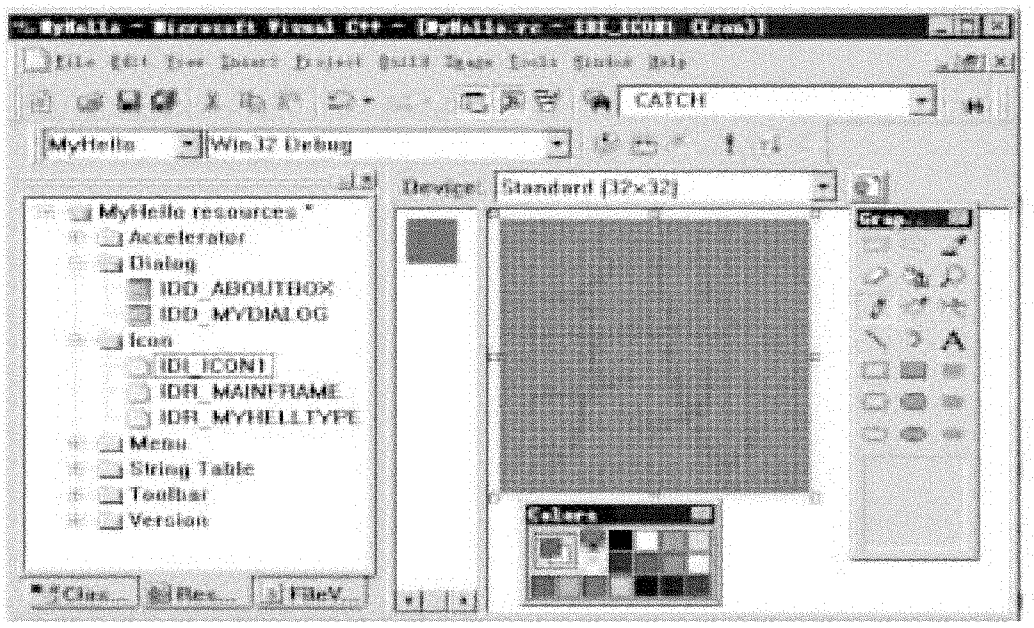



图 2.19　图标资源编辑器

### 2.3.4　菜单(Menu)

菜单设计是界面设计的重要组成部分。设计工作分两类:一是首先在工程中插入新的菜单,然后再编辑添加菜单项;二是在某菜单中新添一个菜单项。

**1.** 在工程中插入新的菜单

实现方法是在“Workspace”窗口的“Resource View”标签中右击“Menu”项,在激活的快捷菜单中选择“Insert Menu”选项,新建一个 ID 为“IDR _ MENU1”的菜单,如图 2.20 所示。

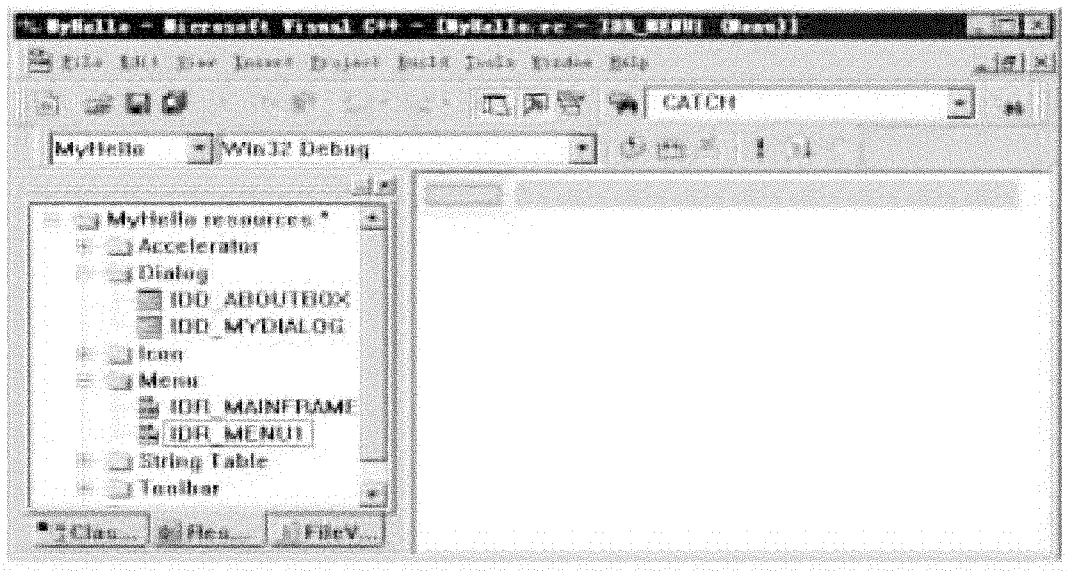



图 2.20　插入新的菜单“IDR _ MENU1”

**2.** 新添一个菜单项

假设在 IDR _ MAINFRAME 菜单的“编辑”菜单下，安添加一个“我的菜单”子菜单项。操作步骤如下：

（1）双击应用框架提供的“IDR _ MAINFRAME”菜单项，在右侧的菜单资源编辑器中，准备新添一个菜单项，如图 2.21 所示。

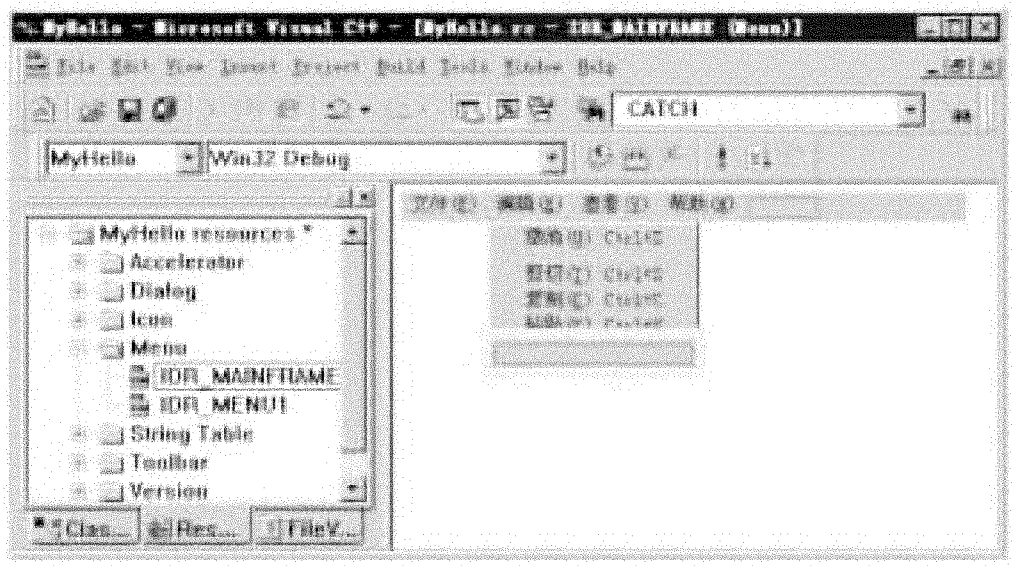



图 2.21 菜单资源编辑器

（2）双击“编辑”菜单下的空菜单项，弹出如图 2.22 所示的“Menu Item Properties”（菜单项属性）对话框，在“ID”文本框中输入新添加菜单项的标识符“ID _ MYMENU”，或选择其对应的事件

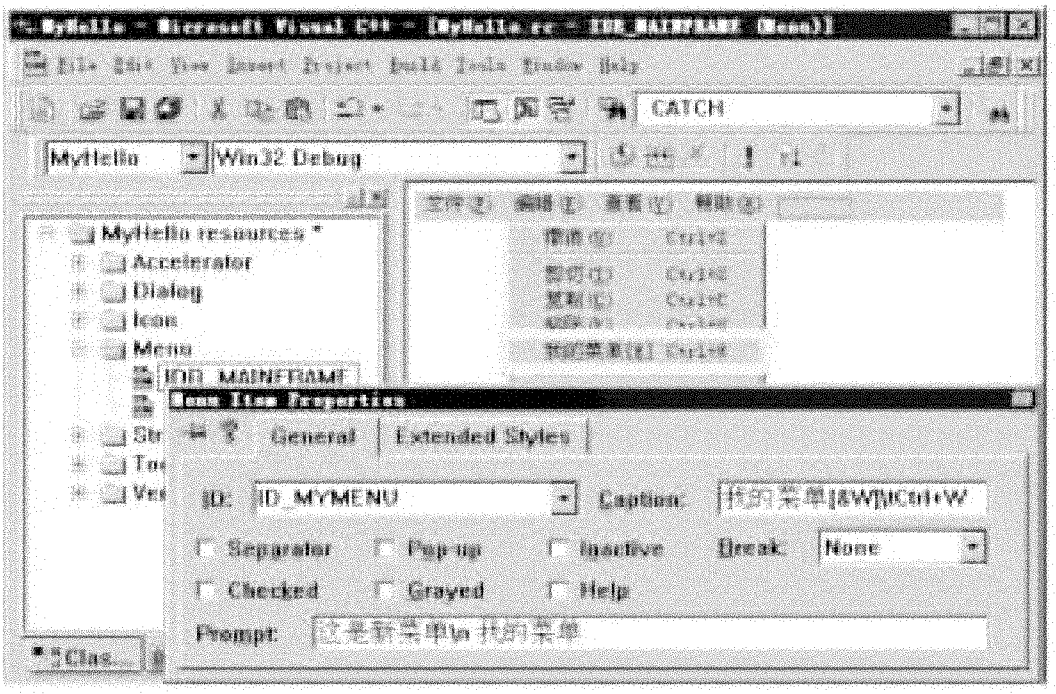



图 2.22 Menu Item Properties”对话框

即可。在“Caption”文本编辑框中输入“我的菜单”。注意在编辑“Caption”文本框的同时，菜单项显示其文本。

要特别注意的是：在图2.22中，“Caption”文本框中含有&符号的字符串“我的菜单[&W]\tCtrl+W”，其意义是在后续字母W标识下划线，并且在程序运行时键入ALT+W(快捷键)执行菜单选项时对应的事件。用&定义的快捷键仅在该菜单激活时有效。类似TAB键格式符后只是说明信息，而无实际操作意义。“Prompt”文本框中的内容是程序运行过程中，当鼠标停在菜单选项时，状态栏中显示的提示信息。

### 2.3.5　字符串表(String Table)

字符串是将应用程序的所有事件的对应的标识号、内部值及其简要说明组织在一起的表。在“Workspace”窗口中的“Resource View”标签中，双击“String Table”中的选项，打开字符串表资源编辑器，如图2.23所示，在其中可以编辑有关内容。

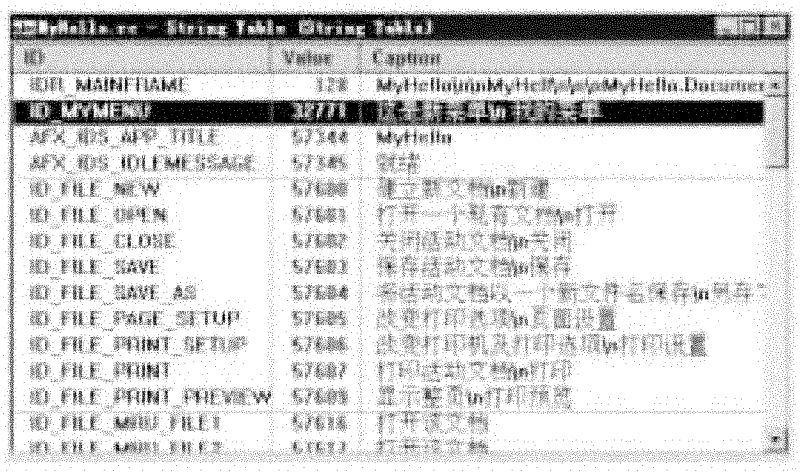



图2.23　字符串表

例如，表中可查到前面新添加的菜单项“ID_MYMENU”的标题“这是新菜单\n我的菜单”等内容(即显示前面编辑的“ID_MYMENU”对话框)。

双击字符串表中ID为“ID_MYMENU”，弹出“String Properties”对话框，如图2.24所示。在该对话框中，可以编辑、修改“Caption”文本框中的内容。

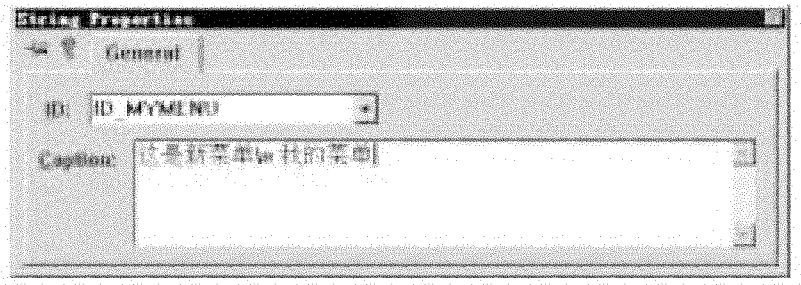



图2.24　“String Properies”对话框

### 2.3.6 工具栏(Toolbar)

在界面设计中,经常需要使用工具栏编辑器设计工具栏。下面为前面新增菜单项“我的菜单”设计一个工具栏按钮,整个一作分两步完成。

**1.** 编辑工具栏按钮

在“Workspace”窗口的资源“ResourceView”标签列表中,双击“Toolbar”下的“IDR_MAINFRAME”项,Visual C++6.0 显示如图 2.25 所示的工具资源编辑器,同时打开“Graphics”和“Colors”绘画工具箱。设计工具栏与设计位图、图标类似,区别在于一个工具栏全部按钮图标保存在一个位图文件中,大小可控且有相关的功能。

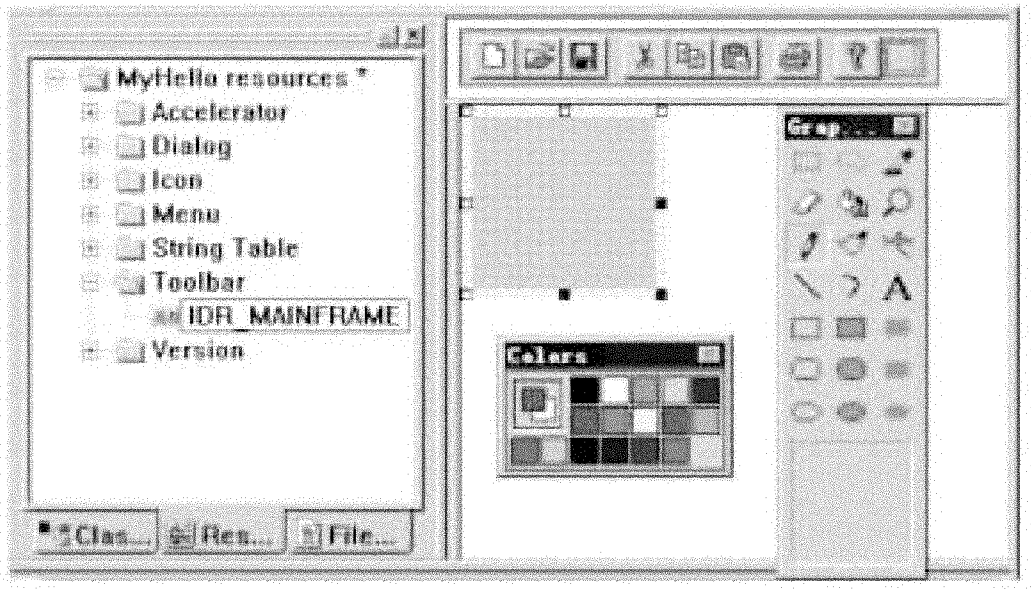



图 2.25 工具资源编辑器

**2.** 设置工具按钮

双击工具栏上新编辑好的按钮(或选择对应按钮后按 Enter 键),Visual C++6.0 显示“Toolbar Button Properties”对话框,如图 2.26 所示。在“ID”列表框中选择“ID_MYMENU”,在“Prompt”文本框中输入“这是新菜单\n 我的菜单”,至此,工具栏按钮设计完毕。


图 2.26 “Toolbar Button Properties”对话框

特别说明：

在“Prompt”文本框中，其中“\n”之前的内容为程序运行时状态栏上给出的该按钮的提示信息；“\n”之后的内容为鼠标光标移至该按钮时光标下方给出的提示信息。

## 习　题　二

1. Visual C++的集成开发环境提供了哪些资源编辑器？
2. 在工程MyHello中，为新增菜单项“我的菜单”添加“Ctrl + N”快捷健。
3. 编辑修改“IDD _ ABOUTBOX”对话框资源，再编译、连接运行程序，观察结果。

# 第3章

# MFC的消息和命令

本章导读

Windows程序设计是基于事件驱动的，Windows对消息有一套完善的严格定义，并在其产生时将其发送给所有相关的应用程序，这些消息用于驱动应用程序运行以实现一定的功能。

本章将通过学习Windows操作系统的消息和MCF的消息处理机制，使读者掌握如下内容：

- Windows消息种类
- MFC的消息处理机制
- 鼠标消息处理方法
- 键盘消息处理方法
- 自定义消息处理方法

---

## 3.1 Windows操作系统的消息

在Windows程序设计中，消息是一个极为重要的概念，用户通过窗口界面的各种操作最后转化为发送到程序中的对象的各种消息，然后由Windows系统交由相应的函数处理。

### 3.1.1 Windows消息的发送和接收

一条Windows消息由消息号、字参数和长整型参数3部分组成，消息号是一条消息的标识，它同时表达了消息的意义和来源，字参数即wParam标识，长整型参数即lParam参数，它们共同给出了消息号的值。

消息被发送到其所对应的窗口，也即是说只有窗口才能接收Windows发送的各种消息。对于MFC来说，只有主框架类CMainFrame和视图类CView及其派生类才能接收消息。

此外，应用程序也能够发送消息，作为用户，也可以发送和接收自己定义的消息以便满足应用的需求。Windows提供两个发送消息的函数：

- PostMessage()

- SendMessage()

其中函数PostMessage()在发送完消息之后就返回了，而不去理会该消息是否已被处理；而SendMessage()函数在发送完消息后，直到等到该消息被处理完毕后才返回。因此在使用时要根据具体情况而定。

### 3.1.2　MFC的消息处理机制

Windows操作系统是通过格式化的消息(Messages)在应用程序中通信的，每个事件发生后，Windows就将它转化为一条消息，判断这条消息应该由哪个窗口来处理，然后将该消息发往该窗口，并交由该窗口的拥有程序去处理，这就是Windows的消息处理机制。而将一条消息翻译并交由相应窗口去处理的过程就是消息映射。

消息映射是Visual C++编程中极为重要的部分。正因为有了这样一种机制，编程工作才会变得如此简单。有了它，用户只关心如何处理一个消息(也就是一个事件，因为事件是产生消息的原因)，而其余的工作只是将这些消息与其相应的处理函数相对应。

消息映射在应用程序中是如何实现的呢？Visual C++的消息映射包括两方面的内容，下面以工程MyHello为例来说明。

(1) 在类的定义中(在MyHelloView.h中)加上一行宏调用：

```
    DECLARE_MESSAGE_MAP()
class CMyHelloView : public CView
{
    protected: // create from serialization only
    CMyHelloView();
    DECLARE_DYNCREATE(CMyHelloView)

    // Attributes
    public:
    CMyHelloDoc * GetDocument();
    ……
    // Generated message map functions
protected:
     //{{AFX_MSG(CMyHelloView)
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()//宏调用
};
```

(2) 在类的实现文件(MyHelloView.cpp)中加上消息映射表：

```
BEGIN_MESSAGE_MAP(类名,父类名)
……
//{{AFX_MSG_MAP(类名)
消息映射入口项
//}}AFX_MSG_MAP
// Standard printing commands
……
END_MESSAGE_MAP()
```

即：

```
BEGIN_MESSAGE_MAP(CMyHelloView, CView)
//{{AFX_MSG_MAP(CMyHelloView)
//}}AFX_MSG_MAP
// Standard printing commands
ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

MyHelloView.h 中有 DECLARE_MESSAGE_MAP()宏，它的作用是用来声明在该类中将使用消息映射来向函数映射消息。在视图类的实现文件中，可以看到有 BEGIN_MESSAGE_MAP()宏，该宏的作用是在该类中开始消息映射。

接下来看到的是 ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)宏，该宏是具体实现所需要的消息映射。它有两个参数：第一个 ID_FILE_PRINT，这是一个 ID。第二个参数 CView::OnFilePrint，这是为 ID 添加的消息处理函数名称。这样，通过 ON_COMMAND 宏就将菜单与相应的消息处理函数进行了连接，也就完成了该 ID 的消息映射。

END_MESSAGE_MAP()宏是结束消息映射。

值得注意的是 ON_COMMAND 宏只用来处理由菜单、工具栏和加速键产生的消息，其他的消息处理可能由别的宏来处理的，详细情况请参考表 3.1。

表 **3.1** 几种常见的消息映射宏

| 消息映射宏 | 说　明 |
|---|---|
| ON_COMMAND | 指出由哪个函数处理指定的命令消息 |
| ON_CONTROL | 指出由哪个函数处理指定的控制消息 |
| ON_MESSAGE | 指出由哪个函数处理指定的用户自定义消息 |
| ON_REGISTERED_MESSAGE | 指出由哪个函数处理指定的已登记用户定义消息 |
| ON_UPDATE_COMMAND_UI | 指出由哪个函数处理指定的用户接口更新命令消息 |

为了更形象地说明 Windows 的消息接收和发送机制，给出如图 3.1 所示的示意图：左边是应用程序间的消息传递示意图，即程序要通过操作系统向另一个应用程序发消息；右边是用户与应用程序的交互，用户也要通过操作系统来发送消息。

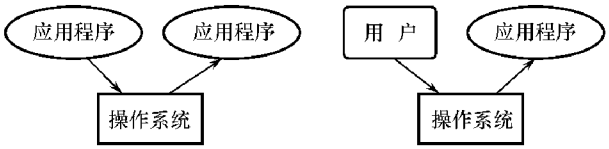



图 3.1 Windows 消息处理机制

特别说明：

在类中，消息响应函数都是类的成员函数。要对消息进行响应，就要定义该消息的响应函

数。在类中，添加一个消息响应函数包括下面3个内容：

(1)在类的定义中，加入该消息处理成员函数的函数原型(函数声明)；

(2)在类的消息映射表中加入相应的消息映射入口项；

(3)在类的实现中加入该消息处理成员函数的函数体。

消息处理函数的原型前面要以关键字afx_msg打头。当用ClassWizard给某个类添加一个消息处理函数时，它会自动加入这3部分内容，只不过它生成的函数体只有一个框架，需要用户添加具体的代码。

### 3.1.3　Windows的消息分类

事件驱动围绕着消息的产生与处理展开，消息是一种报告有关事件发生的通知。事件驱动是靠消息循环机制来实现的。

消息类似于DOS下的用户输入，但比DOS的输入来源要广，Windows应用程序的消息来源有以下4种不同类型：

**1.** 标准**Windows**消息

标准Windows消息由窗口和视图来处理，它通常包括WM前缀，例如WM_CREATE，WM_PAINT等，它们又可以分为鼠标、键盘和窗口消息三类，主要消息如表3.2所示。

表**3.2**　鼠标、键盘和窗口消息的类型

| 消息类型 | 消息标识(ID) | 说明(产生消息的事件) |
|---|---|---|
| 鼠标消息 | WM_MOUSEMOVE | 鼠标移动时发送该消息 |
| | WM_LBUTTONDOWN | 鼠标左键被按下时发送该消息 |
| | WM_LBUTTONUP | 鼠标左键被释放时发送该消息 |
| | WM_LBUTTONDBLCLK | 鼠标左键被双击时发送该消息 |
| | WM_RBUTTONDOWN | 鼠标右键被按下时发送该消息 |
| | WM_RBUTTONUP | 鼠标右键被释放时发送该消息 |
| | WM_RBUTTONDBLCLK | 鼠标右键被双击时发送该消息 |
| 键盘消息 | WM_CHAR | 将一次单击键翻译成一个非系统字符时，发送该消息 |
| | WM_KEYDOWN | 按下一个非系统键时，发送该消息 |
| | WM_KEYUP | 释放一个非系统键时，发送该消息 |
| 窗口消息 | WM_CREATE | 生成一个窗口时发送该消息 |
| | WM_DESTROY | 销毁一个窗口时发送该消息 |
| | WM_CLOSE | 关闭一个窗口时发送该消息 |
| | WM_SIZE | 改变窗口大小时发送该消息 |
| | WM_MOVE | 移动一个窗口时发送该消息 |
| | WM_PAINT | 重画窗口工作区时发送该消息 |

**2.** 控件通告消息

控件通告消息包括按下按钮或输入字符等事件的消息。同标准Windows消息一样，控件通告消息由窗口和视图处理。例如，当用户对编辑控件中的文本作出修改后，编辑控件向其父窗口

发送的 WM _ COMMAND 消息中包含 EN _ CHANGE 控件通告码。窗口的消息处理函数将对该通告消息作出合适的处理，例如接收输入到控件中的文本。

**3.** 命令消息

命令消息也以 WM _ COMMAND 为消息名，在消息中包含有命令的标识符(ID)，以区分具体的命令。命令消息的来源是以下 3 种用户接口对象：

(1) 菜单　用户选择某菜单，要产生相应的命令消息；

(2) 工具栏　用户按下工具栏按钮产生相应的命令消息；

(3) 加速键　用户在键盘上按下了定义的加速键也产生相应的命令消息。

命令消息与其他消息不同，它可被更广泛的对象(如文档、文档模块、应用程序模块等)处理。

**4.** 自定义消息

用户可以自定义消息，在应用程序中主动发出，一般用于应用程序的某一部分内部处理。

## 3.2 Windows 程序框架

在 Windows 中，每个应用程序都是基于事件和消息的，而且包含一个主事件标准 Windows 消息循环：

```
while(GetMessage(&msg,…))
      {
         TranslateMessage(&msg); // 转换键盘消息
         DispatchMessage(&msg); // 分派标准消息
      } // end while
```

其中的 GetMessage( )函数是从时间序列中获得下一个消息，将消息存储在 msg 结构中，TranslateMessage( )函数是为了将键盘消息转化，DispatchMessage( )函数会将消息传给窗口函数去处理(即调用 WinProc( )进行消息处理)。

该循环持续反复检测是否有用户事件发生。每次检测到一个用户事件，程序就对之作出响应。这些事件包括移动鼠标、按键、单击或双击按钮等。当 Windows 接收到这些事件后，就会产生一些相应的消息。应用程序接到这些消息后，按每个消息的专门用途，产生一系列的执行动作。这些动作称之为响应。响应包括重画窗口、改变窗口大小、关闭窗口、打开一个位图、调用一个动态库等。

所以，在 Windows 环境下的编程，都必须恰当地处理消息。消息在 Windows 环境下是按一定顺序发送的。这种消息队列在不同机器上有不同的顺序。当消息队列中的消息依次传送到应用程序框架内部时，应用程序先对其进行分析判断，遇到相应的消息时才处理，否则将跳过这条消息，继续运行。

Windows 程序的执行顺序取决于事件发生的顺序，程序的执行顺序是由顺序产生的消息驱动的，但消息的产生往往并不要求有次序之分。程序员可针对消息类型编写程序以处理接收到的消息，或发出其他消息以驱动其他程序。消息驱动的 Windows 程序框架如图 3.2 所示。

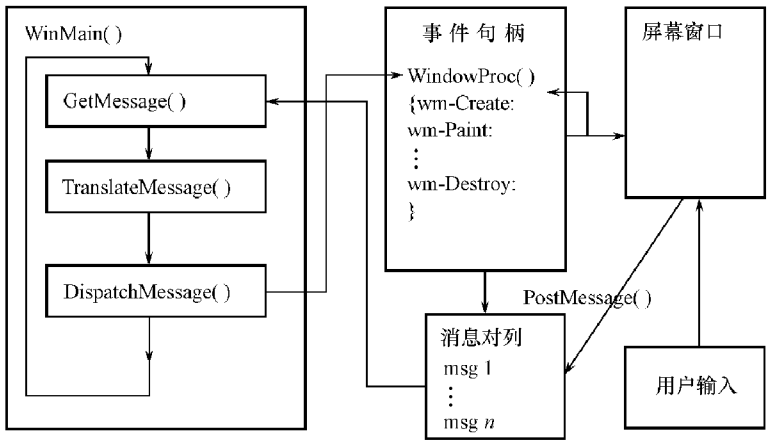



图3.2　消息驱动的Windows程序框架

## 3.3　鼠标消息处理实例

鼠标是Windows操作系统中最重要的输入工具之一。相应地，在MFC编程中，伴随鼠标输入，Windows将会产生相应的消息。

例如，当用户按下并释放鼠标左键时，Windows就会产生WM _ LBUTTONDOWN和WM _ LBUTTONUP两个消息，如果需要，用户只要对这些消息编写相应的响应函数以完成相应的功能，否则大部分的消息，Windows系统均有默认的处理。

### 3.3.1　鼠标消息处理程序

当用户对鼠标进行操作时，会产生相应的消息，系统将把此消息发送到对应的窗口。常见的几种鼠标消息如表3.2所示。

下面对MyHello程序进行改造，添加响应鼠标输入功能。如图3.3所示，当按下鼠标左键或释放或移动鼠标时，程序主窗口将显示相应的动作和屏幕位置。

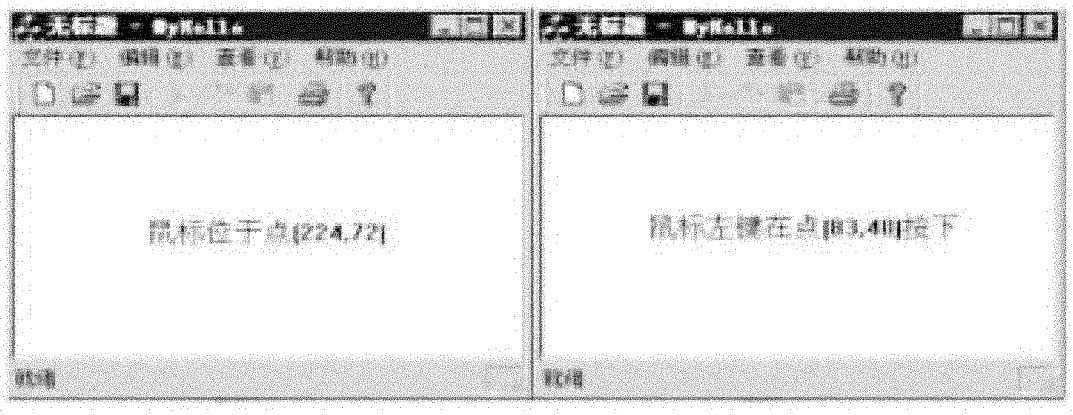


图3.3　响应鼠标输入的运行结果

### 3.3.2 声明视图类的数据成员

为了记录用户操作鼠标的方式和位置，需定义一个变量存储，因此，在视图类中添加一数据成员。

```
class CMyHelloView : public CView
{
……
protected:
   CString m_MousePoint;// 存储鼠标的方式和位置
// Generated message map functions
protected:
……
};
```

在视图类的构造函数中初始化。

```
CMyHelloView::CMyHelloView()
{
     // TODO: add construction code here
      m_MousePoint = " ";
}
```

### 3.3.3 修改屏幕重画函数 OnDraw()

将原显示字符串:“Hello，我们开始 VC++ 编程了!”的语句注释掉。添加显示存储鼠标动作和位置等信息的变量 m_MousePoint 的语句。

```
void CMyHelloView::OnDraw(CDC* pDC)
{
     CMyHelloDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     // TODO: add draw code for native data here
     //pDC->TextOut(100,80,"Hello，我们开始 VC++ 编程了!");
     pDC->TextOut(100,100, m_MousePoint);
}
```

### 3.3.4 添加鼠标消息 WM_LBUTTONDOWN 响应函数

(1) 从“View”菜单中选择“ClassWizard”菜单项，Visual C++ 显示一个“MFC ClassWizard”对话框。

(2) 在“MFC ClassWizard”对话框中，选择“Message Maps(消息映射)”标签，如图 3.4 所示。进行如下设置：

```
Class Name:  CMyHelloView;
Object IDs: CMyHelloView;
```

Messages:　　WM_LBUTTONDOWN;

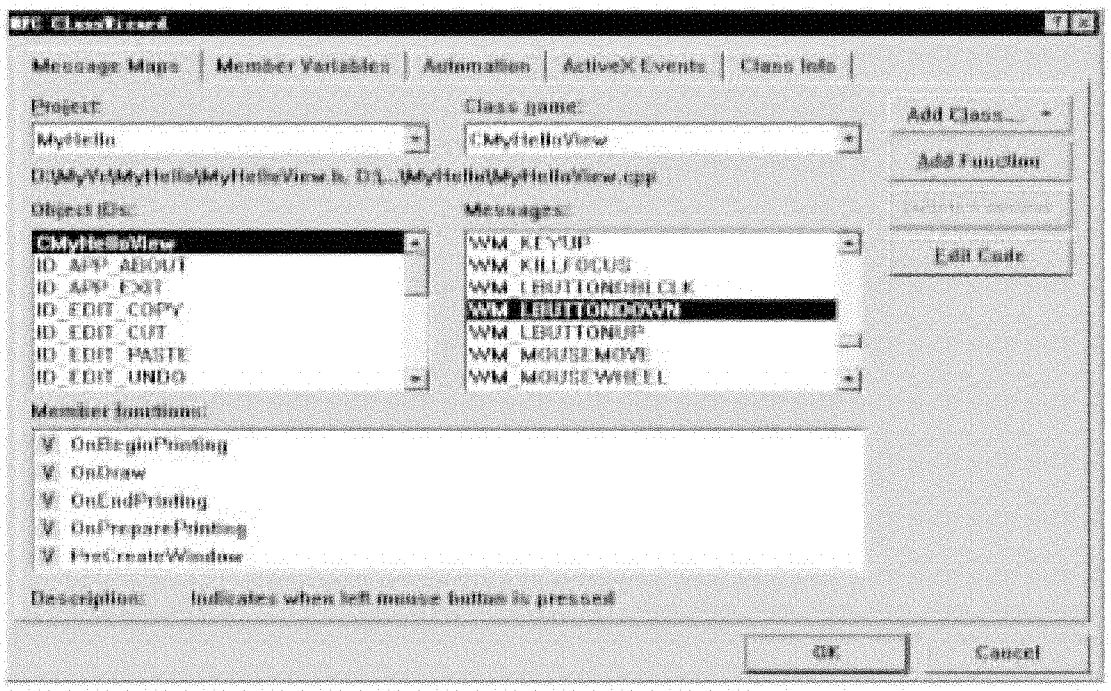



图3.4　“MFC ClassWizard”对话框

（3）单击“Add Function”按钮，就在CMyHelloView类添加了鼠标消息WM_LBUTTONDOWN的响应函数，如图3.5所示。

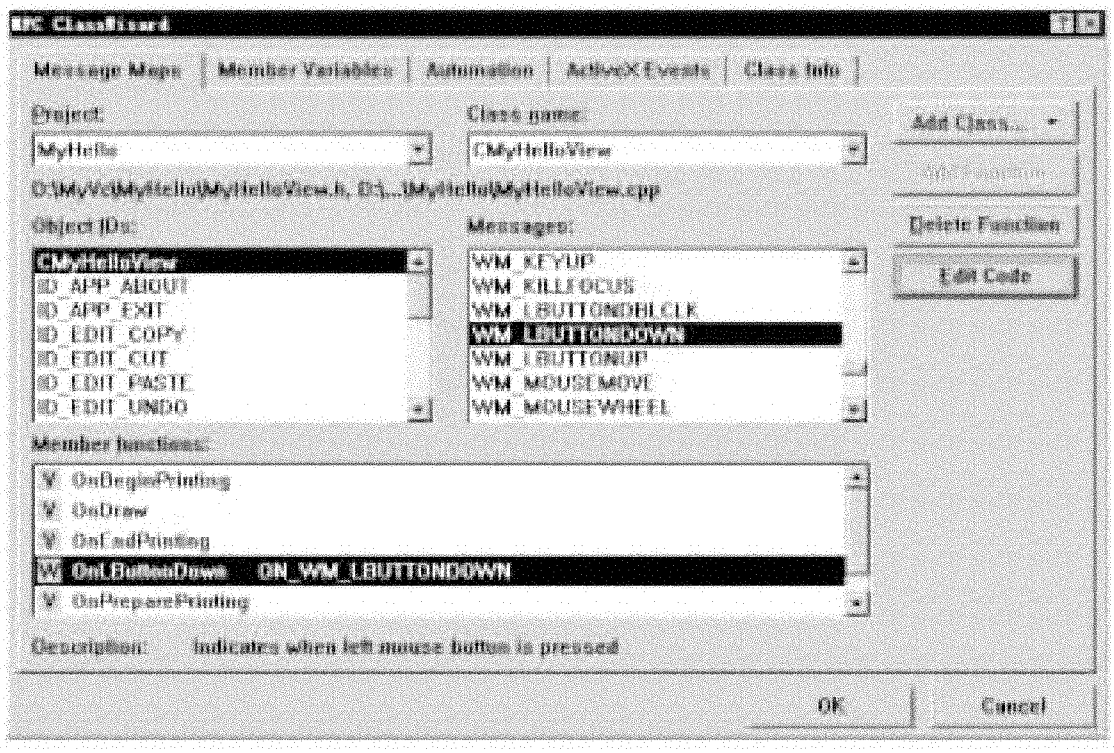



图3.5　添加了鼠标消息WM_LBUTTONDOWN的响应函数

用同样的方法，在 CMyHelloView 类中，添加了鼠标消息 WM _ LBUTTONUP 和 WM _ MOUSEMOVE 的响应函数。

从而，ClassWizard 自动为应用程序做了以下三件事：

（1）在 CMyHelloView 类中添加了 3 个成员方法，即在 MyHelloView.h 中添加了 3 个响应函数的原型说明。

```
// MyHelloView.h : interface of the CMyHelloView class
class CMyHelloView : public CView
{
protected: // create from serialization only
     CMyHelloView();
     DECLARE_DYNCREATE(CMyHelloView)
......

// Generated message map functions
protected:
     //{{AFX_MSG(CMyHelloView)
     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()
};
```

（2）在 MyHelloView.cpp 中添加了 3 个消息映射。

```
BEGIN_MESSAGE_MAP(CMyHelloView, CView)
   //{{AFX_MSG_MAP(CMyHelloView)
   ON_WM_LBUTTONDOWN()
   ON_WM_LBUTTONUP()
   ON_WM_MOUSEMOVE()
   //}}AFX_MSG_MAP
   // Standard printing commands
   ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
   ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

（3）在 MyHelloView.cpp 中添加了 3 个响应函数的空函数体。

```
void CMyHelloView::OnLButtonDown(UINT nFlags, CPoint point)
{
     // TODO: Add your message handler code here and/or call default

     CView::OnLButtonDown(nFlags, point);
}

void CMyHelloView::OnLButtonUp(UINT nFlags, CPoint point)
{
     // TODO: Add your message handler code here and/or call default
```

```
    CView::OnLButtonUp(nFlags, point);
}

void CMyHelloView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default

    CView::OnMouseMove(nFlags, point);
}
```

### 3.3.5　编写消息响应函数代码

在“MFC ClassWizard”对话框中，选中要编辑的成员函数，单击“MFC ClassWizard”对话框中的“Edit Code”按钮，鼠标将定位其响应函数（或直接在 MyHelloView.cpp 程序中，定位要编辑的函数），编写记录鼠标动作和位置信息的代码。

```
void CMyHelloView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    m_MousePoint.Format("鼠标左键在点(%d,%d)按下", point.x,point.y);
    Invalidate();
    CView::OnLButtonDown(nFlags, point);
}

void CMyHelloView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    m_MousePoint = "鼠标左键被释放";
    Invalidate();
    CView::OnLButtonUp(nFlags, point);
}

void CMyHelloView::OnMouseMove(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    m_MousePoint.Format("鼠标位于点(%d,%d)", point.x,point.y);
    Invalidate();
    CView::OnMouseMove(nFlags, point);
}
```

### 3.3.6　查看结果

（1）选择“Build”菜单中的“Build MyHello.exe”菜单项，Visual C++ 就会编译并连接 MyHello.exe 程序。

（2）选择“Build”菜单中的“Execute MyHello.exe”菜单项，Visual C++ 就会执行 MyHello.exe 程序，如图 3.3 所示的 MyHello.exe 程序主窗口也随之出现。

（3）进行按下鼠标左键、释放和移动鼠标等操作，可以达到设计的功能。

### 3.3.7 技术要点

在响应鼠标消息函数中，语句：

```
Invalidate();
```

的作用是使系统让用户区无效，从而调用重画函数 OnDraw( )。所以用户只要在其中编写显示 m _ MousePoint 的语句即可。

## 3.4 键盘消息处理实例

除鼠标外，键盘也是 Windows 操作系统中最重要的输入工具之一。在 Windows 的编程环境中，伴随着键盘的输入将产生相应的 Windows 键盘消息。

例如，用户按下和释放一个键盘键时，就将产生 WM _ KEYDOWN 和 WM _ KEYUP 两个消息，而且 Windows 将此键翻译成 ASCII 码后又将产生 WM _ CHAR 消息，用户只要对这些消息中的一部分编写相应的响应函数即可，对其中大部分的消息，Windows 系统均有默认的处理方法。

### 3.4.1 键盘消息处理程序

当用户对键盘进行操作时，会产生相应的消息，系统将把此消息发送到对应的窗口。常见的几种键盘消息如表 3.2 所示。

下面，对 MyHello 程序再次进行改造，添加响应键盘的字符输入。程序运行效果如图 3.6 所示。

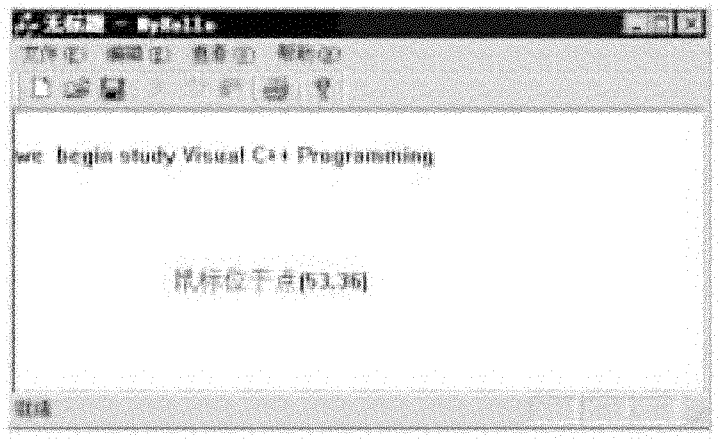



图 3.6 响应键盘字符输入

### 3.4.2 声明视图类的数据成员

为了记录用户输入的回车次数和存储在当前行输入的字符，需定义一个整型变量存储回车次数，CString 型变量存储当前行输入的字符，所以，在视图类中添加 2 个数据成员：

```
class CMyHelloView : public CView
    {
……
protected:
     CString m_MousePoint;
     int m_nLine;// 存储回车次数
     CString m_strDisplay;// 存储当前行输入的字符

// Generated message map functions
protected:
……
};
```

在视图类的构造函数中对这两数据成员进行初始化：

```
CMyHelloView::CMyHelloView()
{
     // TODO: add construction code here
     m_MousePoint = " ";
     m_nLine = 0;
}
```

### 3.4.3　添加键盘消息 WM_CHAR 响应函数

为健盘消息添加响应函数的步骤如下：

（1）从“View”菜单中选择“ClassWizard”菜单项，Visual C++显示一个“MFC ClassWizard”对话框。

（2）在“MFC ClassWizard”对话框中选择“Message Maps(消息映射)”标签。

（3）在对话框中进行如下设置：

```
Class name:   CMyHelloView;
Objiect IDs: CMyHelloView;
Messages:     WM_CHAR;
```

（4）单击“Add Function”按钮，就在 CMyHelloView 类添加了键盘消息 WM_CHAR 的响应函数，如图 3.7 所示。

同样，如同添加响应鼠标消息函数一样，ClassWizard 自动为应用程序做了以下三件事：

（1）在 CMyHelloView 类中添加了一个成员方法，即在 MyHelloView.h 中添加了一个响应函数的原型说明。

```
class CMyHelloView : public CView
{
……
protected:
     //{{AFX_MSG(CMyHelloView)
     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
```

```
        afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

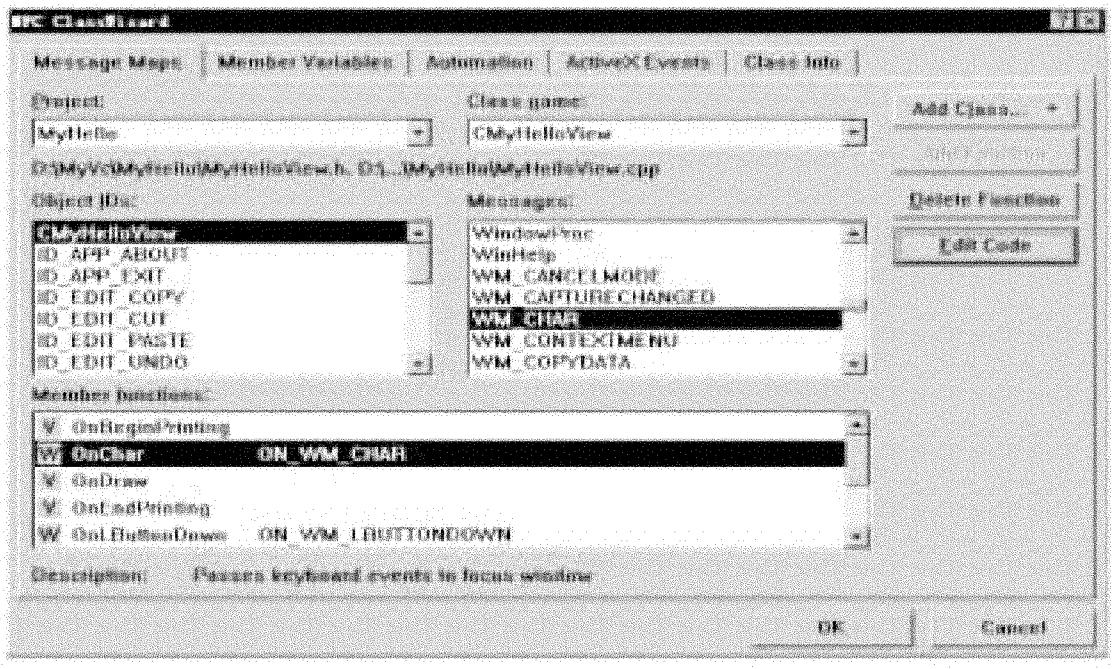



图 3.7 添加了键盘消息 WM_CHAR 的响应函数

（2）在 MyHelloView.cpp 中添加一个消息映射。

```
BEGIN_MESSAGE_MAP(CMyHelloView, CView)
    //{{AFX_MSG_MAP(CMyHelloView)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    ON_WM_CHAR()
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

（3）在 MyHelloView.cpp 中添加一个响应函数的空函数体。

```
void CMyHelloView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default

    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

### 3.4.4　编辑消息响应函数

单击“MFC ClassWizard”对话框中的“Edit Code”按钮，定位其消息响应函数，编写在视图窗口显示输入字符的代码。

```
void CMyHelloView::OnChar(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    if(nChar == VK_RETURN)//如按下回车键
    {
        m_strDisplay.Empty(); //清空变量
        m_nLine++;    //行数加一
    }
    else {
        m_strDisplay += nChar; //将输入的字符添加到变量 m_strDisplay 的尾端
    }
    CClientDC dc(this);
    dc.TextOut(0,m_nLine*20,m_strDisplay);
    CView::OnChar(nChar, nRepCnt, nFlags);
}
```

### 3.4.5　查看结果

（1）选择“Build”菜单中的“Build MyHello.exe”菜单项，Visual C++就会编译并连接MyHello.exe程序。

（2）选择“Build”菜单中的“Execute MyHello.exe”菜单项，Visual C++就会执行MyHello.exe程序，如图3.6所示的MyHello.exe程序主窗口也随之出现。

（3）单击键盘字符，字符会显示在窗口内。

（4）单击回车键，再次输入字符时，将在下一行的最左端开始显示。

## 3.5　定时器消息处理实例

定时器能够以固定的时间间隔产生WM_TIMER事件，它不同于硬时钟中断，它是用软件实现的。在程序中使用定时器时要设置一个WM_TIMER事件发生的频率，例如，如果把定时器的时间间隔设置为500 ms(0.5 s)，那么每500 ms就会产生一个WM_TIMER事件。

### 3.5.1　定时器程序

如果要使程序运行时每隔一定的时间间隔，发出“滴答”声，这可通过定时器来实现。该功能的实现可以分解为两步：

（1）创建一个定时器，使之在规定的时间间隔，发送特定的消息。

(2) 在消息响应函数中,编写发“滴答”声音的代码。

### 3.5.2 安装定时器

要在程序中使用定时器,必须为程序安装一个定时器。安装定时器的方法如下:

(1) 选择“View”菜单的“ClassWizard”菜单项。

(2) 在“MFC ClassWizard”对话框的“Message Maps”标签中做如下设置:

```
Class name:   CMyHelloView
Object IDs:   CMyHelloView
Messages:     OnInitalUpdate
```

(3) 单击“Add Function”按钮,添加 OnInitalUpdate()函数,如图 3.8 所示。

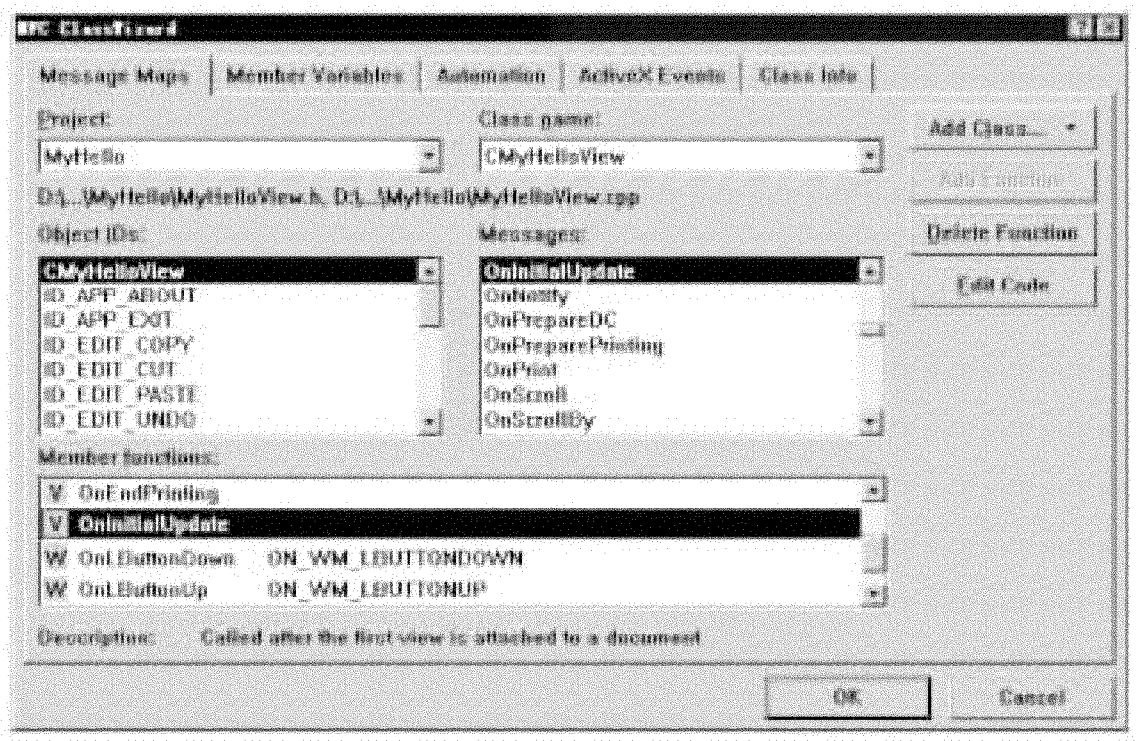



图 3.8 添加 OnInitalUpdate()函数

(4) 在 OnInitalUpdate 函数中,添加安装定时器代码。

```
void CMyHelloView::OnInitialUpdate()
{
        CView::OnInitialUpdate();
        // TODO: Add your specialized code here and/or call the base class
        SetTimer(1,500,NULL);// 安装定时器
}
```

### 3.5.3 清除定时器

定时器安装成功后,可以通过调用 KillTimer( )函数清除定时器。例如,可以在程序退出时清

除定时器，操作如下：

（1）选择“View”菜单的“ClassWizard”菜单项。

（2）在“MFC ClassWizard”对话框的“Message Maps”标签中做如下设置：

```
Class name:   CMyHelloView
Object IDs:   CMyHelloView
Messages:     WM_DESTROY
```

（3）单击“Add Function”按钮，添加OnDestroy()函数，如图3.9所示。

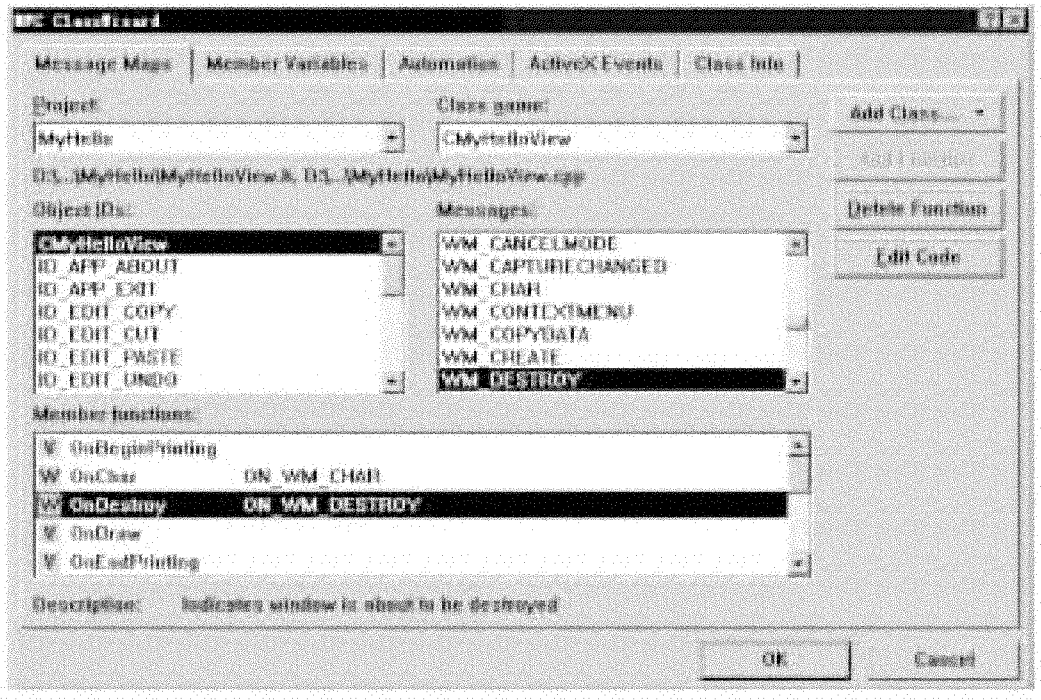



图3.9　添加WM_DESTROY消息响应函数

（4）单击“Edit Code”按钮，Visual C++就会打开MyHelloView.cpp文件，等待用户编辑OnDestroy()函数。

（5）在OnDestroy()中输入清除定时器的代码。

```
void CMyHelloView::OnDestroy()
{
    CView::OnDestroy();
    // TODO: Add your message handler code here
    KillTimer(1);// 清除定时器
}
```

### 3.5.4　添加定时器消息WM_TIMER响应函数

为了测试定时器，必须先添加WM_TIMER消息的响应函数，步骤如下：

（1）选择“View”菜单的“ClassWizard”菜单项。

（2）在“MFC ClassWizard”对话框的“Message Maps”标签中进行如下设置：

```
Class name:   CMyHelloView
Object IDs:   CMyHelloView
Messages:     WM_TIMER
```

（3）单击“Add Function”按钮，添加 OnTimer()函数，如图 3.10 所示。

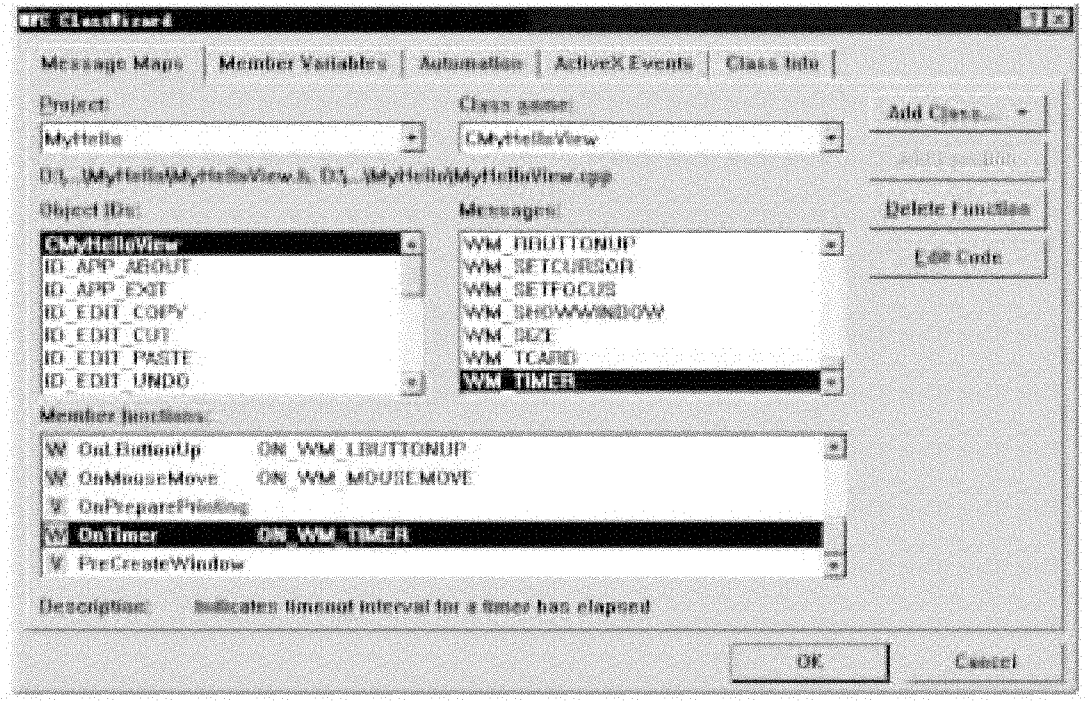



图 3.10 添加 WM_TIMER 消息的响应函数

（4）单击“Edit Code”按钮，Visual C++ 就会打开 MyHelloView.cpp 文件，等待用户编辑 OnTimer()函数。

（5）在 OnTimer()中输入响应定时器消息的代码。

```
void CMyHelloView::OnTimer(UINT nIDEvent)
{
    // TODO: Add your message handler code here and/or call default
    MessageBeep(-1);
    CView::OnTimer(nIDEvent);
}
```

### 3.5.5 查看结果

编译、连接并执行 MyHello 程序，出现如图 3.6 所示的界面，程序运行时会在一定的时间间隔，重复发出“滴答”声音。

### 3.5.6 技术要点

#### 1. SetTimer()函数

安装定时器函数 CWnd::SetTimer()的函数原型为：

```
UINT SetTimer( UINT nIDEvent, UINT nElapse,
               void (CALLBACK EXPORT* lpfnTimer)(HWND, UINT, UINT, DWORD) );
```

其中参数

nIDEvent：定时器的ID，标识定时器的非零序号。作为清除该定时器标识。

nElapse：指定WM_TIMER事件发生的时间间隔，以毫秒为单位。

lpfnTimer：指定一个用于响应WM_TIMER消息的处理函数地址，如果设置为NULL，则WM_TIMER消息放在消息队列中，由窗口对象处理。

如果安装成功，则返回非零。

**2．KillTimer()函数**

清除定时器函数CWnd::KillTimer()的函数原型为

```
BOOL KillTimer( int nIDEvent );
```

其中的参数是安装定时器时所指定的定时器的ID。

当程序退出时，必须调用KillTimer()函数以清除定时器。退出窗口的方法有两种，第一种方法是用户单击窗口右上角的关闭按钮，但在结束之前会产生WM_DESTORY事件，所以OnDestroy()函数中的KillTimer()函数就会首先执行；第二种方法是用户用鼠标点击系统菜单中的Exit关闭菜单项，此时也会产生WM_DESTORY事件。

## 3.6　自定义消息处理实例

用户可以自定义消息，在应用程序中主动发出，这种消息一般用于应用程序的某一部分内部处理。例如，在MyHello.exe程序中，添加如下功能：

当用户单击光标上移键时，程序发送用户自定义消息，在对应的消息响应函数中，弹出消息对话框，显示消息发送成功，如图3.11所示。

图3.11　发送自定义消息运行情况

### 3.6.1 基本知识

**1.** 定义用户消息的 **ID**(标识符)

在头文件中用如下语句定义用户消息的标识符:WM _ USERMSG

```
const WM _ USERMSG = WM _ USER + 100
```

其中:

WM _ USER 是由 Windows 定义的,它是第一个有效的用户消息。因为程序的其他部分也会使用用户消息,故将新的用户消息 WM _ USERMSG 设置为 WM _ USER + 100。

**2.** 用户消息的发送

Windows 提供发送消息的函数::PostMessage()函数,函数 PostMessage()的功能是将消息插入消息队列并返回,由处理线程处理,而队列中的消息通过 GetMessage()或 PeekMessage()函数获取。PostMessage()函数的原型为:

```
BOOL PostMessage(
          HWND hWnd,              // handle of destination window
          UINT Msg,               // message to post
          WPARAM wParam,          // first message parameter
          LPARAM lParam           // second message parameter
);
```

其中 PostMessage()的 4 个参数分别是接收消息的窗口的句柄、消息的 ID、消息的 WPARAM 和 LPARAM 参数。

### 3.6.2 定义用户消息和消息响应函数

**1.** 定义用户消息

将下面的语句加入到 MyHelloView.h 中 CMyHelloView 类声明的上面。

```
// MyHelloView.h : interface of the CMyHelloView class
……
const WM _ USERMSG = WM _ USER + 100;

class CMyHelloView : public CView
{
protected: // create from serialization only
     CMyHelloView();
     ……
};
```

**2.** 声明用户消息响应函数

将用户消息响应函数 OnMyFunction()作为视图类的公有段成员函数,也就是在 MyHelloView.h 文件中,添加 void OnMyFunction()。

```
// MyHelloView.h : interface of the CMyHelloView class
……
const WM_USERMSG = WM_USER + 100;
class CMyHelloView : public CView
{
protected: // create from serialization only
     CMyHelloView();
     DECLARE_DYNCREATE(CMyHelloView)

// Attributes
public:
     CMyHelloDoc* GetDocument();

// Operations
public:
     void OnMyFunction();//用户消息响应函数
// Overrides
……
protected:
     //{{AFX_MSG(CMyHelloView)
     afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
     afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
     afx_msg void OnMouseMove(UINT nFlags, CPoint point);
     afx_msg void OnChar(UINT nChar, UINT nRepCnt, UINT nFlags);
     afx_msg void OnDestroy();
     afx_msg void OnTimer(UINT nIDEvent);
     //}}AFX_MSG
     DECLARE_MESSAGE_MAP()
};
```

**3.** 定义用户消息的响应函数

在 CMyHelloView.cpp 中，添加 OnMyFunction()函数体，并编写代码。为简单起见，添加弹出消息对话框代码。

```
void CMyHelloView::OnMyFunction()
{
    MessageBox("恭喜你,消息发送成功!");
}
```

### 3.6.3　添加消息映射

在 MyHelloView.cpp 文件的类的消息映射中加入有关的语句，位置紧挨在}}AFX_MSG_MAP之后。

```
BEGIN_MESSAGE_MAP(CMyHelloView, CView)
   //{{AFX_MSG_MAP(CMyHelloView)
   ON_WM_LBUTTONDOWN()
   ON_WM_LBUTTONUP()
   ON_WM_MOUSEMOVE()
```

```
    ON_WM_CHAR()
    ON_WM_DESTROY()
    ON_WM_TIMER()
    //}}AFX_MSG_MAP
    ON_MESSAGE(WM_USERMSG,OnMyFunction)
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CView::OnFilePrintPreview)
END_MESSAGE_MAP()
```

### 3.6.4 编写程序代码

**1. 添加 WM_KEYDOWN 消息响应函数**

当用户单击光标上移键时，将产生 WM_KEYDOWN 消息。添加键盘 WM_KEYDOWN 消息响应函数的操作步骤如下：

（1）选择“View”菜单的“ClassWizard”菜单项。

（2）在“MFC ClassWizard”对话框的“Message Maps”标签中进行如下设置：

```
Class name:   CMyHelloView
Object IDs:   CMyHelloView
Messages:     WM_KEYDOWN
```

（3）单击“Add Function”按钮，添加 OnKeyDown()函数，如图 3.12 所示。

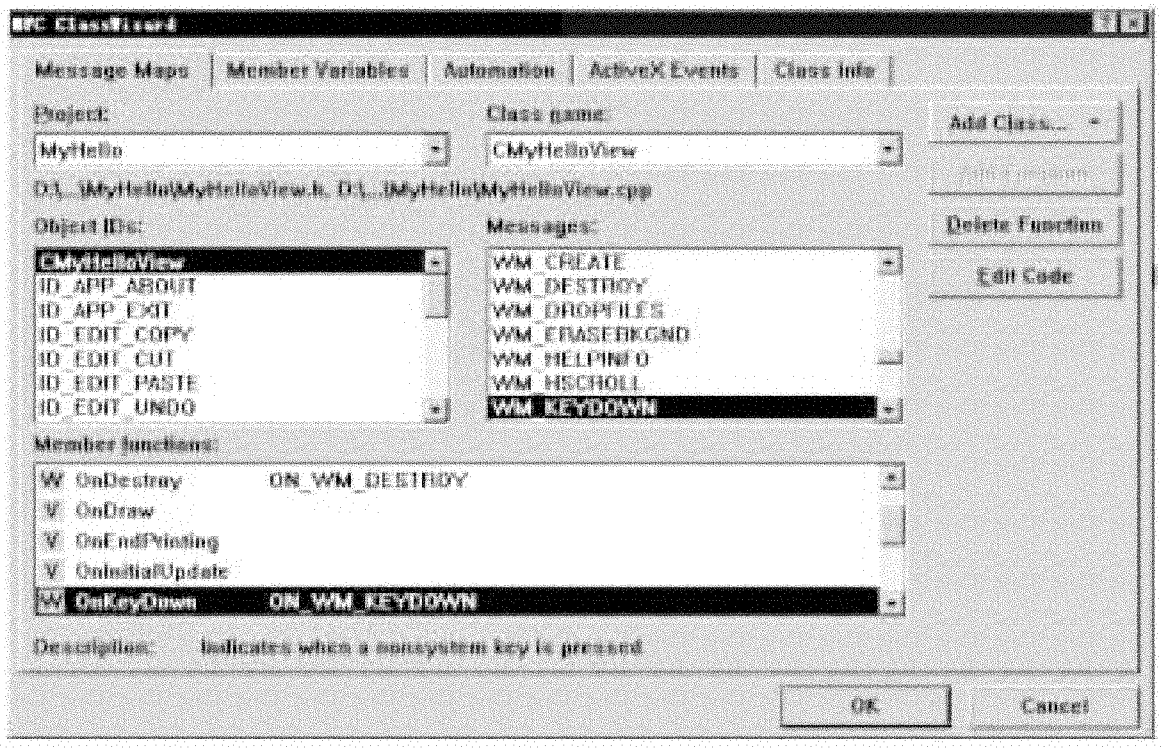



图 3.12 添加 WM_KEYDOWN 消息响应函数

（4）单击“Edit Code”按钮，Visual C++就会打开MyHelloView.cpp文件，等待用户编辑OnKeyDown()函数：

```
void CMyHelloView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default

    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

**2.** 编写发送自定义消息代码

在OnKeyDown()中输入发送自定义消息的代码。

```
void CMyHelloView::OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)
{
    // TODO: Add your message handler code here and/or call default
    HWND hWnd = GetSafeHwnd();
    if(nChar == VK_UP){
            ::PostMessage(hWnd,WM_USERMSG,0,0);
        return;
    }

    CView::OnKeyDown(nChar, nRepCnt, nFlags);
}
```

**3.** 查看结果

编译、连接并执行MyHello程序，单击光标上移键时，程序将弹出消息对话框，如图3.11所示。

### 3.6.5　技术要点

在函数OnKeyDown(UINT nChar, UINT nRepCnt, UINT nFlags)中，第一个参数nChar为所按的键的字符代码值，第二个参数nRepCnt为重复击键的次数，第三个参数nFlags表示扫描码和键转换前后的状态。表3.3是nFlags的功能描述表。

表3.3　**nFlags**功能描述表

| 对应位 | 含　义 |
|---|---|
| 0～15 | 表示某键被重复按下的次数 |
| 16～23 | 表示扫描码，依赖于键盘 |
| 24 | 若同时按下扩展键，则置位 |
| 25～28 | 由Windows保留 |
| 29 | 若按下Alt键，则置位 |
| 30 | 表示先前键的状态，之前是按下则置位 |
| 31 | 表明键的转化换状态，若键已松开则置位 |

在函数 OnKeyDown()中，代码：

```
if(nChar == VK_UP){
        ::PostMessage(hWnd,WM_USERMSG,0,0);
    return;
    }
```

中的 VK_UP 是光标上移键的虚拟键。虚拟键以 VK_为前缀，对虚拟键处理时需要它们的参数，表 3.4 给出了一些常用的虚拟键的信息。

表 **3.4** 常用的虚拟键盘码

| 虚拟键名称 | 对应的键盘键 |
|---|---|
| VK_UP | 光标上移 |
| VK_DOWN | 光标下移 |
| VK_LEFT | 光标左移 |
| VK_RIGHT | 光标右移 |
| VK_HOME | Home |
| VK_END | End |
| VK_PRIOR | PageUp |
| VK_NEXT | PageDown |
| VK_RETURN | 回车 |

# 习 题 三

1. 常用的键盘消息有哪些？鼠标消息又有哪些？
2. 改进 MyHello 程序功能，在字符输入显示处添加光标显示。
3. 改进 MyHello 添加 BackSpace 键的删除功能。
4. 改进 MyHello 程序功能，添加光标上、下、左、右移动功能，即使程序具有全屏幕输入。

# 第4章

# 对话框与常用控件

## 本章导读

对话框是实现人机交互的重要手段。大多数应用程序来说，经常需要利用对话框以获取用户的输入，或为用户显示信息。

本章通过开发一个计算器和口令对话框程序，使读者掌握以下内容：

- Button 控件、Edit Box 控件和 Static Text 控件的使用方法
- 基于对话框的应用程序的编程技术
- 模式对话框、非模式对话框和通用对话框的设计与调用

---

## 4.1 MyCalculator 程序

图 4.1 所示是 MyCalculator 程序的运行结果。该程序具有一个普通计算器的功能，如加、减、乘、除四则运算、开方、倒数和清零等。

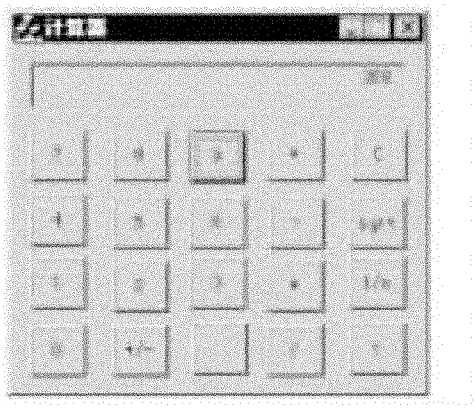


图 4.1 计算器

下面以开发计算器程序 MyCalculator 为目标，学习对话框与常用控件的知识，并以此完成 MyCalculator 程序的设计。

# 4.2 开发 MyCalculator 程序

用 Visual C++6.0 编写 MFC 应用程序，是一种“填空式”的编程方法，一般有 3 个步骤：

（1）创建工程 用 Visual C++6.0 的 MFC AppWizard 生成应用程序的工程文件，也就是创建应用程序的基本框架。

（2）可视化设计 用 Visual C++6.0 自带的工具软件 Winzards，制作 Windows 风格的图形用户界面和各种控件，如矩形按钮、滚动条和单选按钮。

（3）编写程序代码 根据目标程序的要求，用 Visual C++6.0 提供的文本编辑器和 C++ 程序设计语言来编写代码。

## 4.2.1 创建工程

下面是创建 MyCalculator 工程的详细步骤：

（1）启动 Visual C++6.0。从“File”菜单中选择“New”，此时，Visual C++6.0 将显示一个如图 4.2 所示的“New”对话框。

（2）在“New”对话框中选择“Projects”标签，然后指定工程类型 MFC AppWizard[exe]、工程名 MyCalculator 和工程位置 D:\MyVC，如图 4.2 所示。

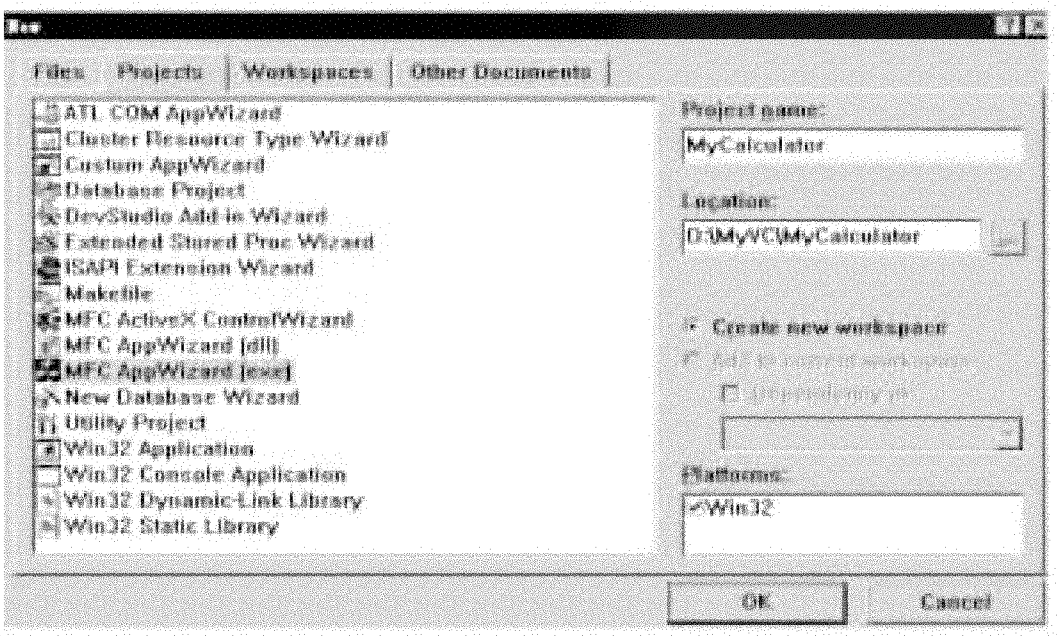



图 4.2 指定工程类型、工程名和工程位置

（3）点击“OK”按钮，弹出“MFC AppWizard - Step 1”对话框。选择“Dialog based”单选按钮，创建一个基于对话框的应用程序，如图 4.3 所示。

（4）单击“Next”按钮，在 MFC AppWizard-Step 2 of 4 对话框中，输入“计算器”作为对话框的标题，其他接受默认设置，如图 4.4 所示。

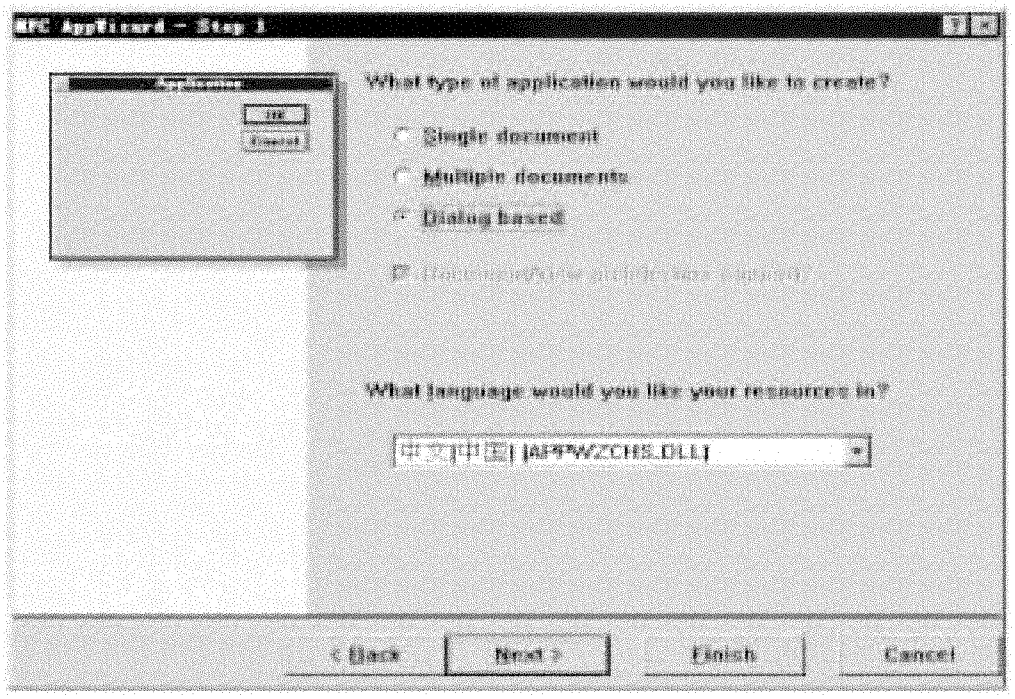



图 4.3　设置“MFC AppWizard – Step 1”对话框

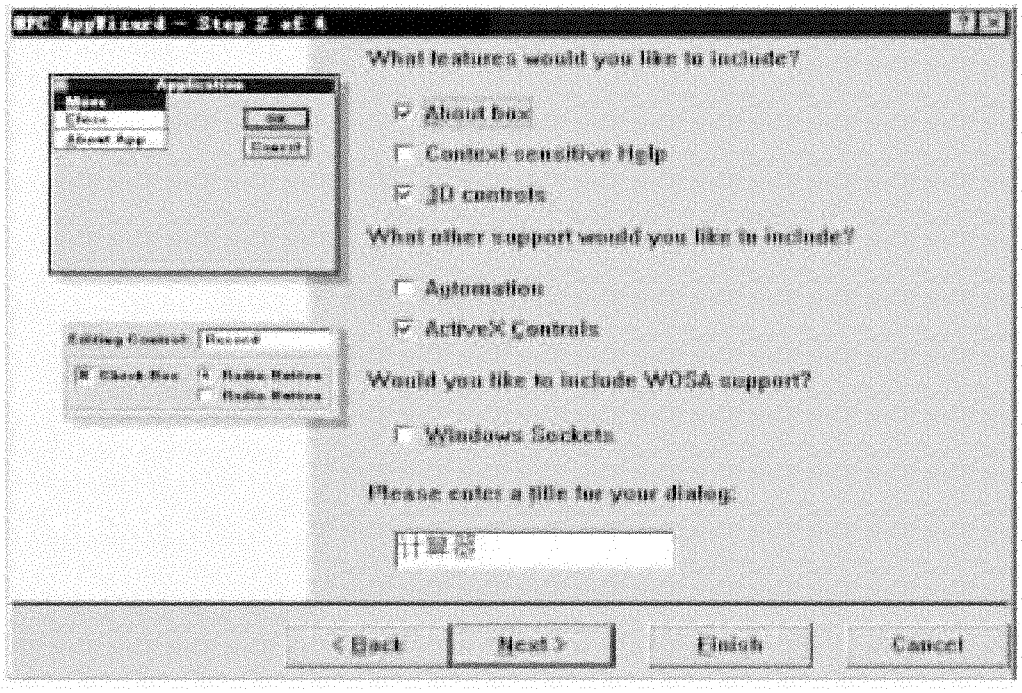



图 4.4　设置“MFC AppWizard – Step 2 of 4”对话框

（5）单击“Next”按钮，显示“MFC AppWizard – Step 3 of 4”对话框，接受默认设置，如图 4.5 所示。

（6）单击“Next”按钮，显示“MFC AppWizard – Step 4 of 4”对话框，这时可看到为应用程序创建了 2 个类：CMyCalculatorApp、CMyCalculatorDlg，如图 4.6 所示。

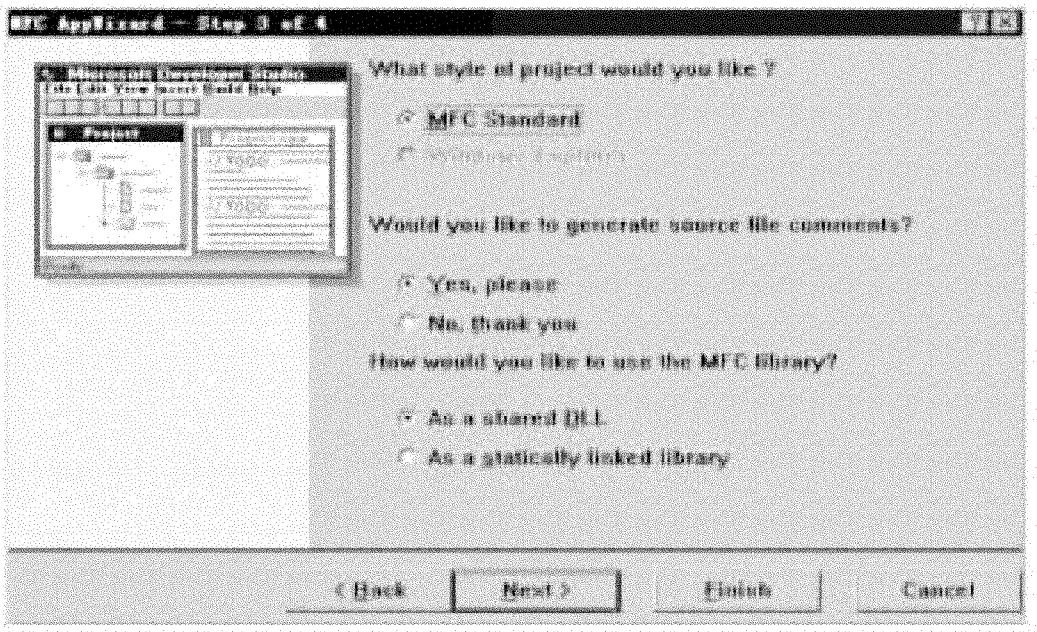



图 4.5 “MFC AppWizard – Step 3 of 4”对话框

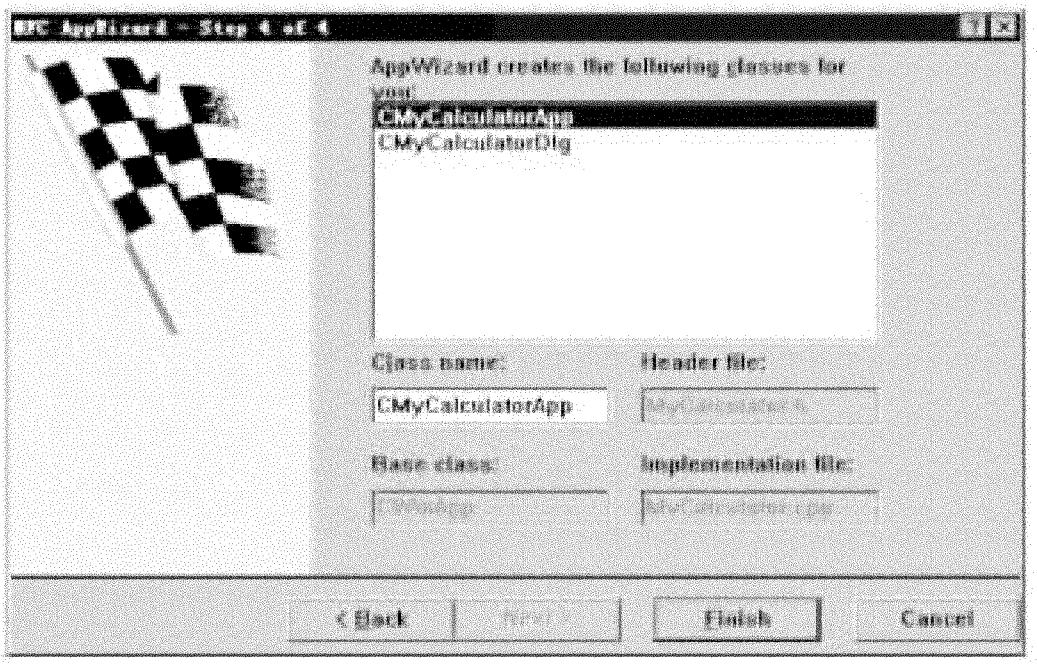



图 4.6 “MFC AppWizard – Step 4 of 4”对话框

(7) 单击“Finish”按钮,此时 Visual C++ 6.0 将显示“NewProject Information”窗口。

(8) 单击“OK”按钮,Visual C++ 6.0 就会创建 MyCalculator 工程以及相关的所有文件。

### 4.2.2 可视化设计

可视化设计,就是用 Visual C++ 6.0 自带的工具软件 Winzards,制作计算器程序界面,即添加

计算器中的数字、运算符等按钮和数字显示编辑框。

操作步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 MyCalculator resources 扩展开，然后再扩展 Dialog，最后，双击“IDD _ MYCALCULATOR _ DIALOG”项，Visual C++ 6.0 显示出处于设计状态的“IDD _ MYCALCULATOR _ DIALOG”对话框。

（2）从“IDD _ MYCALCULATOR _ DIALOG”对话框中删除“OK”和“Cancel”按钮以及 TODO 文本。

（3）将鼠标放置工具栏的任意位置，单击鼠标右键弹出快捷菜单，选择“Controls”复选项（如图 4.7(a)所示），将弹出控件工具箱，如图 4.7(b)所示。

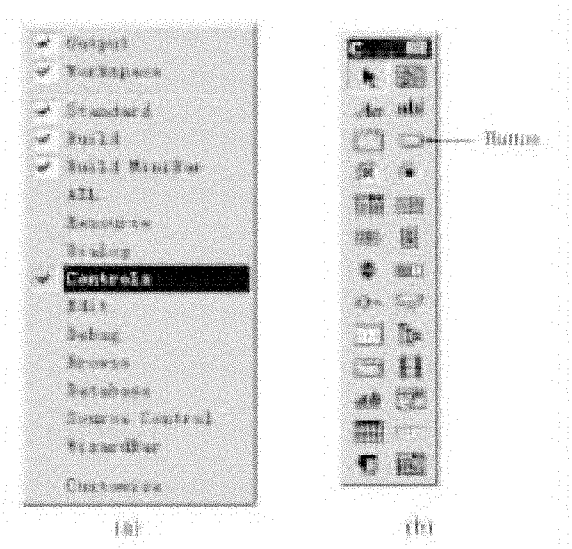



图 4.7　工具栏和工具箱

（4）在控件工具箱中，选中“Button”按钮控件，再单击“IDD _ MYCALCULATOR _ DIALOG”对话框中适当的地方，Visual C++ 6.0 就会把“Button”按钮放置在刚才单击的地方，按钮默认标题是“Button1”，下面把其标题改为“0”。

（5）右击“Button1”按钮，将弹出如图 4.8 所示的快捷菜单。

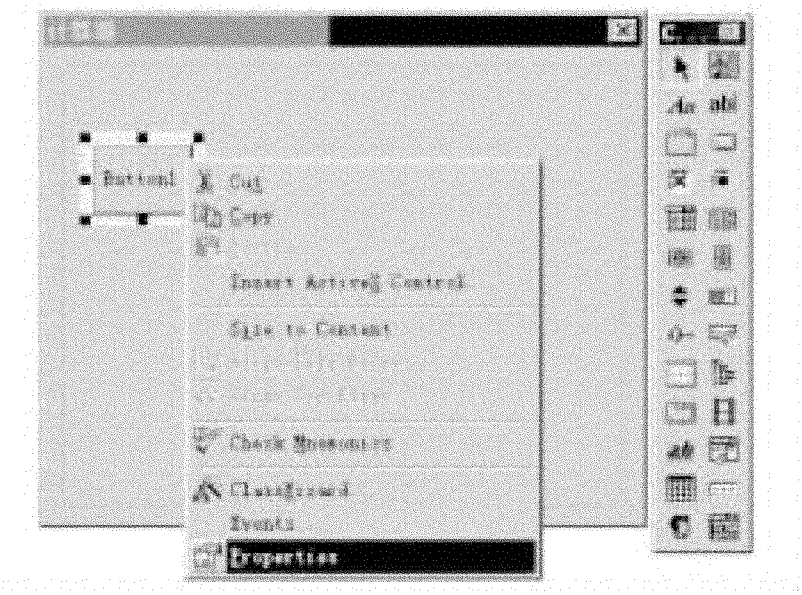



图 4.8　右击“Button1”按钮弹出的快捷菜单

（6）选择弹出菜单的“Properties”（属性）菜单项，将显示“Push Button Properties”对话框。

单击“ID”文本框，把文本“IDC _ BUTTON1”修改为“IDC _ BUTTON0”，同样将“Caption”文本编辑框中“Button1”修改为“0”，如图 4.9 所示。

关闭“Push Button Properties”对话框，并按设计要求移动按钮位置，改变大小，至此为止，数字

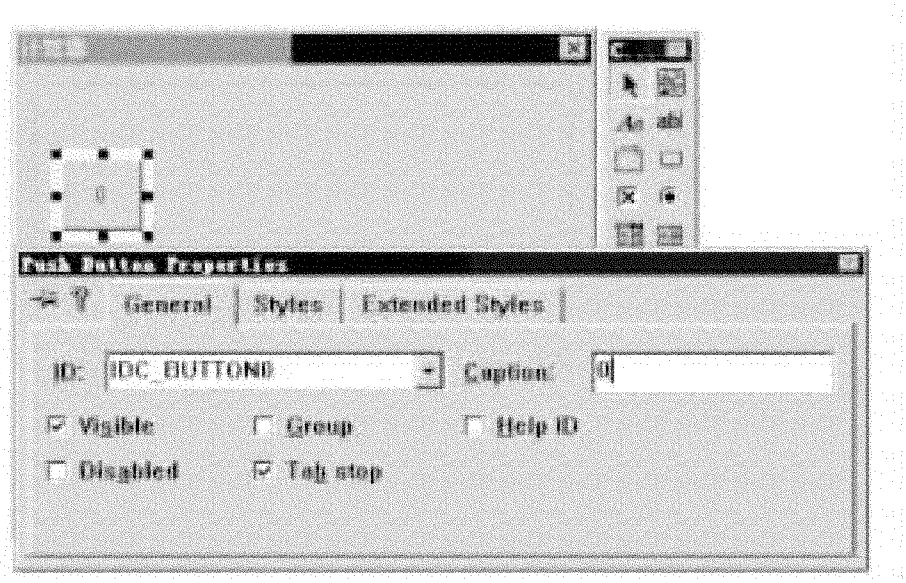



图 4.9 “Push Button Properties”对话框

“0”按钮设计完毕。

用完全类似的方法，根据表 4.1 中的定义，编辑对话框资源，设计完毕后对话框如图 4.10 所示。

表 **4.1** 对话框“**IDD _ MYCALCULATOR _ DIALOG**”中各控件的属性表

| 对 象 | ID 属性 | Caption 属性 |
|---|---|---|
| Button | IDC _ BUTTON0 | 0 |
| Button | IDC _ BUTTON1 | 1 |
| | | |
| Button | IDC _ BUTTON9 | 9 |
| Button | IDC _ BUTTON _ POINT | . |
| Button | IDC _ BUTTON _ SIGN | +/- |
| Button | IDC _ BUTTON _ ADD | + |
| Button | IDC _ BUTTON _ MINUS | - |
| Button | IDC _ BUTTON _ MUTIPLY | * |
| Button | IDC _ BUTTON _ DIV | / |
| Button | IDC _ BUTTON _ CLEAR | C |
| Button | IDC _ BUTTON _ SQRT | sqrt |
| Button | IDC _ BUTTON _ RECIPROCAL | 1/x |
| Button | IDC _ BUTTON _ EQUAL | = |
| Edit Box | IDC _ DISPLAY | Edit |
| | 只读(Read only) | |

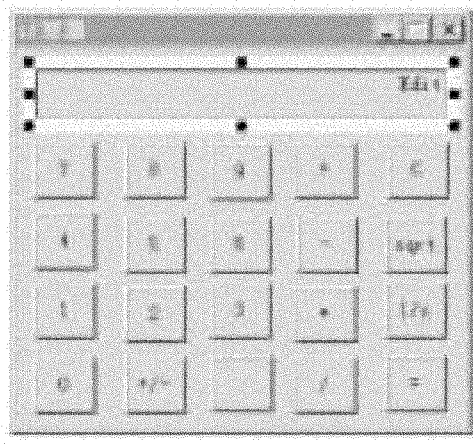


图 4.10　设计完后的“IDD _ MYCALCULATOR _ DIALOG”对话框资源

### 4.2.3　为编辑框“IDC _ DISPLAY”引入变量

为了能够在程序运行过程中，将输入的数据和计算的结果在编辑框上显示，必须为它引入一个变量，从而能够使编辑框以变量的形式出现在程序中。利用函数：

```
UpdateData(true)
```

或

```
UpdateData(false)
```

就可达到目的。

为“IDC _ DISPLAY”编辑框引入变量的步骤如下：

（1）在“View”菜单中选择“ClassWizard”菜单项。

（2）在“MFC ClassWizard”对话框中，选择“Member Variable”标签，如图 4.11 所示。进行如下设置：

```
Class name:  CMyCalculatorDlg
Control IDs: IDC _ DISPLAY
```

（3）单击“Add Variable...”按钮，此时，Visual C++ 将显示一个“Add Member Variable”对话框，如图 4.12 所示，并进行如下设置：

```
Member variable name: m _ display
Category:      Value
Variable type:  CString
```

（4）单击“Add Member Variable”对话框的“OK”按钮，返回到“MFC ClassWizard”对话框，再单击“MFC ClassWizard”对话框的“OK”按钮。于是，Visual C++ 就为编辑框“IDC _ DISPLAY”引入变量 m _ display。

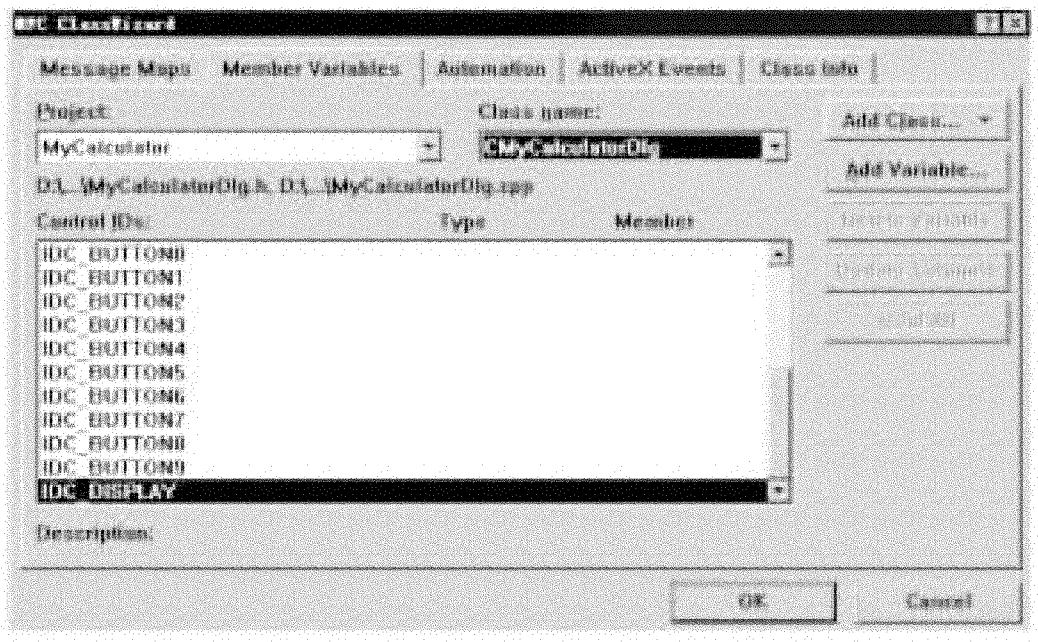



图 4.11 “MFC ClassWizard”对话框

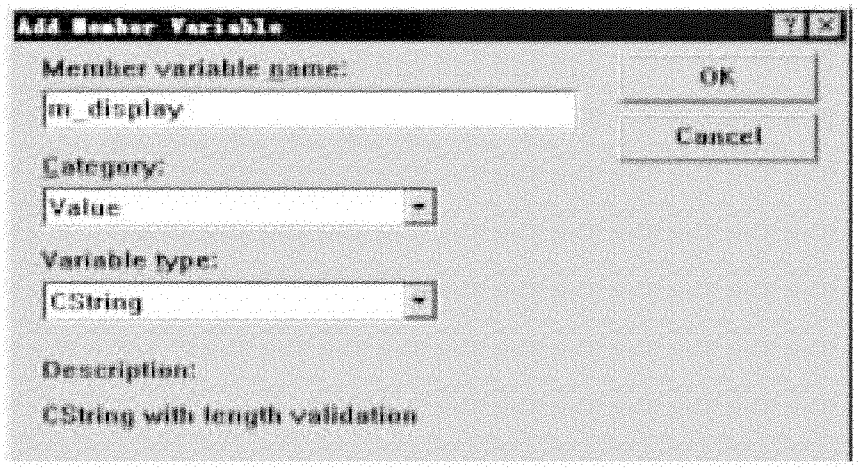



图 4.12 “Add Member Variable”对话框

### 4.2.4 为 CMyCalculatorDlg 类添加数据成员

为了实现计算器功能，需要向 CMyCalculatorDlg 类（在 MyCalculatorDlg.h 文件中）添加变量。变量属性和用途如表 4.2 所示。

表 **4.2** 所用数据成员列表

| 数据类型 | 数据变量名称 | 数据的作用 |
|---|---|---|
| double | m_first | 存储一次运算的第一个操作数以及一次运算的结果 |
| double | m_second | 存储一次运算的第二个操作数 |
| double | m_coff | 存储小数点的系数权值 |
| CString | m_operator | 存储运算操作符 |

**1.** 添加变量 **m _ first**

操作步骤如下：

（1）在工作区窗口“ClassView”标签中，选择“CMyCalculatorDlg”后单击右键，在弹出的快捷菜单中选择“Add Member Variable”，弹出“Add Member Variable”对话框。

（2）在“Add Member Variable”对话框中进行如下编辑和选择，如图 4.13 所示。

```
Variable Type:   double
Variable Name:   m_first
Access:          Protected
```

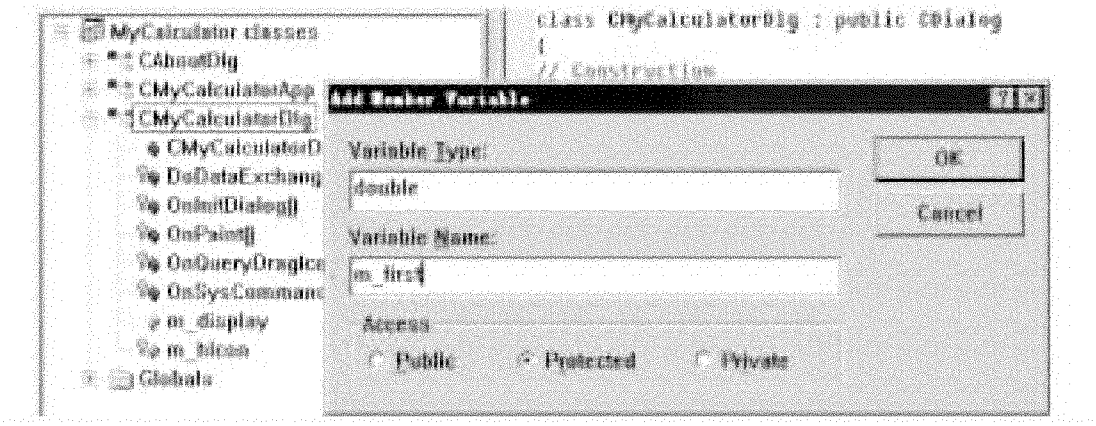



图 4.13　“Add Member Variable”对话框

（3）单击“OK”按钮，则在 CMyCalculatorDlg 类中添加了名为 m _ first 的 double 类型变量，且为保护型数据成员（Protected），用来存储一次运算的第一个数。

重复上述过程，共加入以下 4 个变量：

```
double m_first;
double m_second;
double m_coff;
CString m_operator;
```

说明：向 CMyCalculatorDlg 类添加数据成员变量的另一种简便方法是，在 MyCalculatorDlg.h 文件中，直接用手工方法添加。

**2.** 在构造函数中初始化成员变量

类中的数据成员的初始化是在构造函数中完成，所以必须在构造函数中加入初始化代码（为某些控键引入的变量，自动添加了初始化代码，但用户可以修改）。在 CMyCalculatorDlg 的构造函数中添加初始化代码：

```
CMyCalculatorDlg::CMyCalculatorDlg(CWnd * pParent /* = NULL */)
    : CDialog(CMyCalculatorDlg::IDD, pParent)
{
   //{{AFX_DATA_INIT(CMyCalculatorDlg)
   m_display = _T("0.0");
```

```
    m_first = 0.0;
    m_second= 0.0;
    m_operator=_T("+");
    m_coff = 1.0;
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

### 4.2.5 为 Button 按钮的 BN_CLICKED 事件添加响应函数

用“MFC ClassWizard”为对话框“IDD_MYCALCULATOR_DIALOG”中的所有 Button 按钮的 BN_CLICKED 事件添加处理函数，如表 4.3 所示。

表 4.3 对话框“IDD_MYCALCULATOR_DIALOG”上诸按钮消息的响应函数

| Object IDs | Messages | Member Functions |
|---|---|---|
| IDC_BUTTON1 | BN_CLICKED | OnOnButton1() |
| | | |
| IDC_BUTTON9 | BN_CLICKED | OnOnButton9() |
| IDC_BUTTON_POINT | BN_CLICKED | OnButtonPoint() |
| IDC_BUTTON_SIGN | BN_CLICKED | OnButtonSign() |
| IDC_BUTTON_ADD | BN_CLICKED | OnButtonAdd() |
| IDC_BUTTON_MINUS | BN_CLICKED | OnButtonMinus() |
| IDC_BUTTON_MUTIPLY | BN_CLICKED | OnButtonMutiply() |
| IDC_BUTTON_DIV | BN_CLICKED | OnButtonDiv() |
| IDC_BUTTON_CLEAR | BN_CLICKED | OnButtonClear() |
| IDC_BUTTON_SQRT | BN_CLICKED | OnButtonSqrt() |
| IDC_BUTTON_RECIPROCAL | BN_CLICKED | OnButtonReciprocal() |
| IDC_BUTTON_EQUAL | BN_CLICKED | OnButtonEqual() |

在此，仅以“0”按钮为例，为其 BN_CLICKED 事件添加响应函数，其他按钮完全类似，请读者独立完成。

操作步骤如下：

（1）从“View”菜单中选择“ClassWizard 菜单项”，弹出“MFC ClassWizard”对话框。

（2）选择“Message Maps”标签，并进行如下设置：

```
Class name:   CMyCalculatorDlg
Object IDs:   IDC_BUTTON0
Message:      BN_CLICKED
```

此时，“MFC ClassWizard”对话框如图 4.14 所示。

（4）单击“Add Function...”按钮来增加新函数，在弹出的“Add Member Function”对话框中，接受默认函数名 OnButton0，单击“OK”按钮。此时，“MFC ClassWizard”对话框如图 4.15 所示。

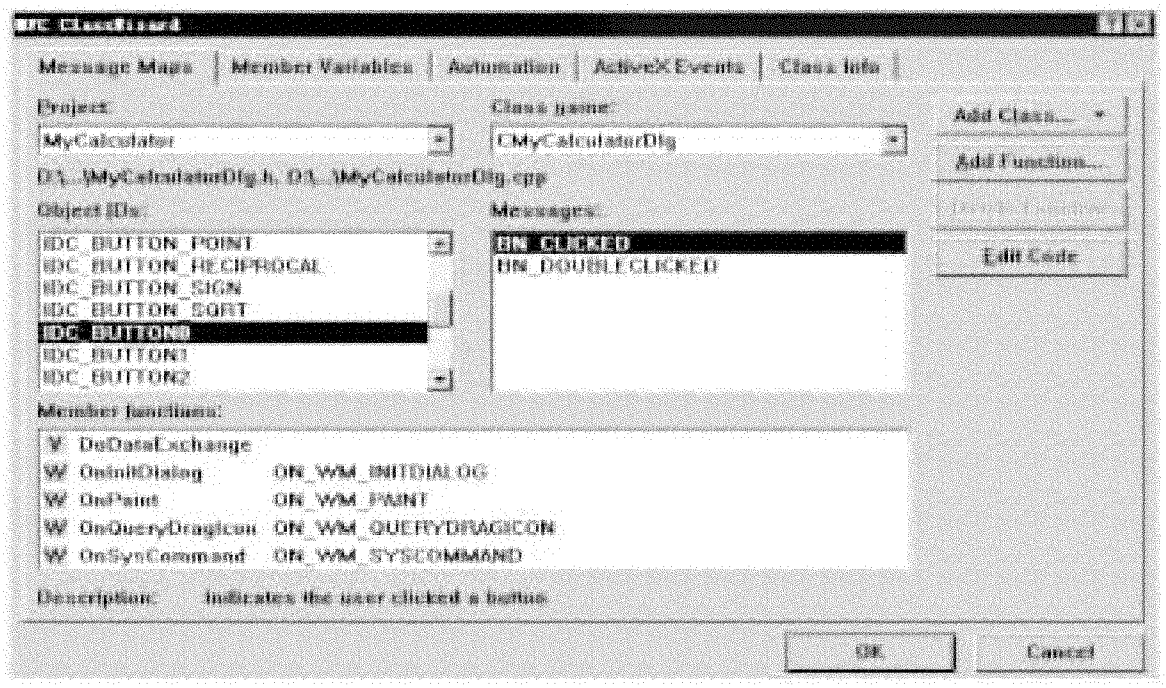



图 4.14　选中“IDC _ BUTTON0”按钮的 BN _ CLICKED 事件

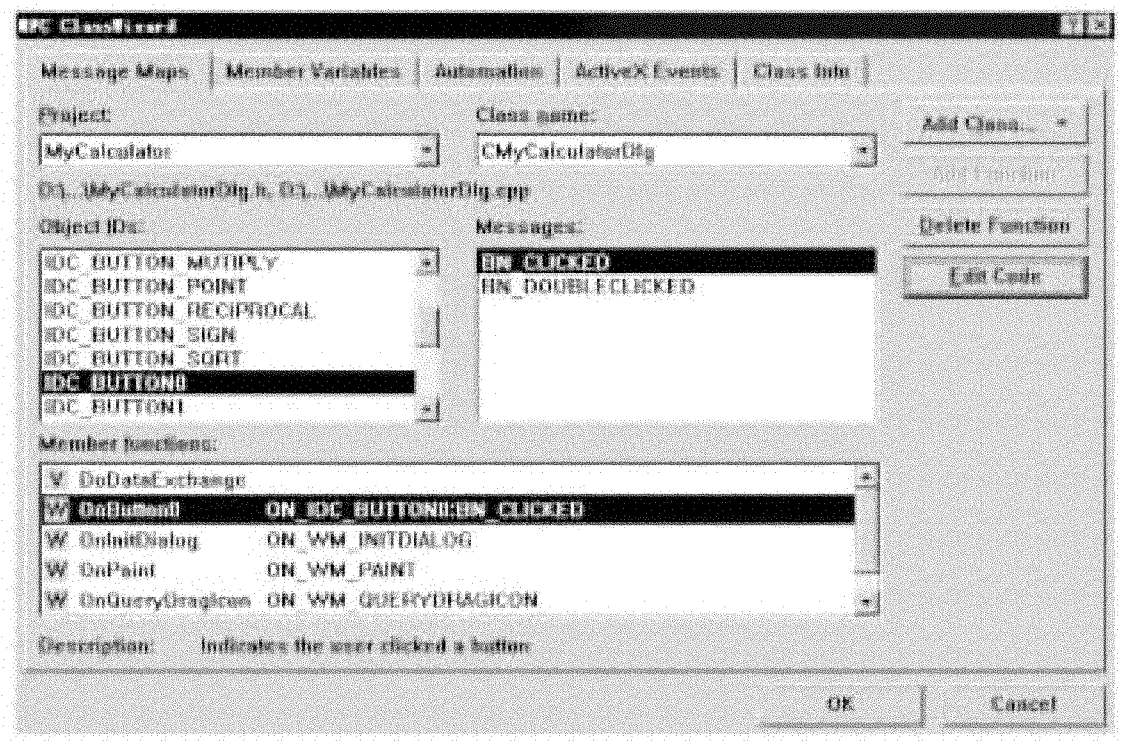



图 4.15　为数字“0”按钮的 BN _ CLICKED 事件添加消息响应函数 OnButton0( )

（5）单击“MFC ClassWizard”对话框的“OK”按钮，就为“IDC _ BUTTON0”按钮添加了 BN _ CLICKED 事件的消息响应函数 OnButton0( )。

```
void CMyCalculatorDlg::OnButton0()
{
}
```

### 4.2.6 编写程序代码

当鼠标单击对话框中某个 Button 按钮(比如“8”)时,导致 Windows 产生 BN _ CLICKED 消息。而要完成某种功能,就要由消息响应函数中的代码来实现。因此,要编写程序代码,具体步骤如下:

**1.** 为数字“**N**”按钮的消息响应函数添加代码

如 4.2.4 的约定,变量 m _ second 是用来存储当前的输入,也就是存储一次运算的第二操作数,m _ coff 存储小数的权值,对于整数和小数的输入要采取不同的处理方式。

算法设计:

(1) 作为整数输入数字 N 时(N = 0,1,2,…,9)

```
m_second = m_second*10 + N;
```

(2) 作为小数输入数字 N 时(N = 0,1,2,…,9)

```
m_second = m_second + N*m_coff;
m_coff  *= 0.1;
```

所以,在数字“8”按钮的消息响应函数中添加如下代码:

```
void CMyCalculatorDlg::OnButton8()
{
    if( m_coff == 1.0)// 作为整数输入
       m_second = m_second*10 + 8;
    else       //作为小数输入
        {m_second = m_second + 8*m_coff;
        m_coff *= 0.1;
        }
    UpdateDisplay(m_second);//更新编辑框的数据显示
}
```

完全类似地,请读者为其他数字按钮的消息响应函数添加代码。

**2.** 为运算符按钮的消息响应函数添加代码

当点击四则运算符(+、-、*、/)时,就是要将前一次数据与当前数据进行运算,作为下次的第一操作数,并在编辑框中显示。

由以下算法描述:

(1) m _ first 与 m _ second 做 m _ operator 运算后→m _ first;

(2) 0→m _ second;

(3) m _ first→m _ display (显示);

(4) 用当前点击的运算符→m _ operator;

(5) 将小数的权值变量 m _ coff 置 1.0 复位。

以点击“+”运算符为例，在消息响应函数中编写如下代码：

```
void CMyCalculatorDlg::OnButtonAdd()
{    Calculate();
     m_operator = "+";
}

void CMyCalculatorDlg::Calculate(void)
{
     switch(m_operator.GetAt(0))
     {
     case '+':
         m_first += m_second;break;
     case '-':
         m_first -= m_second;break;
     case '*':
         m_first *= m_second;break;
     case'/':
          if(fabs(m_second)<= 0.000001)
          { m_display = "除数不能为零";
            UpdateData(false);
            return;
          }
          m_first /= m_second;break;
     }
     m_second = 0.0;
     m_coff = 1.0;
     UpdateDisplay(m_first);//更新编辑框的显示内容
}

void CMyCalculatorDlg::UpdateDisplay(double lVal)
{
     m_display.Format(_T("%f"),lVal);
     int i = m_display.GetLength();
     //格式化输出，将输出结果后的零全部截去
     while(m_display.GetAt(i-1)== '0')
     {m_display.Delete(i-1,1);i--;}
     UpdateData(false); //更新显示编辑框变量 m_display
}
```

其他运算类似，留给读者自己完成。

**3.** 为等号“=”按钮消息响应函数添加代码

当点击等号“=”按钮时，算法设计如下：

（1）m_first 与 m_second 作 m_operator 运算后→m_first；

（2）m_first→m_display（显示）；

（3）0→m_second；

（4）0→m_first；

（5）将小数的权值变量 m_coff 置 1.0；

（6）输入的运算符→m_operator;

```
void CMyCalculatorDlg::OnButtonEqual()
{
   Calculate();
   m_first = 0.0;
   m_operator = " + ";
}
```

**4.** 在 **OnButtonSqrt**()函数中，编写如下代码

```
void CMyCalculatorDlg::OnButtonSqrt()
{
   m_second = sqrt(m_second);
   UpdateDisplay(m_second);
}
```

**5.** 为“+/-”、“.”、“C”和“1/X”按钮的消息响应函数编写代码

```
//“C”按钮的消息响应函数
void CMyCalculatorDlg::OnButtonClear()
{    m_first = 0.0;
     m_second = 0.0;
     m_operator = " + ";
     m_coff = 1.0;
     UpdateDisplay(0.0);
}
//“1/X”按钮的消息响应函数
void CMyCalculatorDlg::OnButtonReciprocal()
{
     if(fabs(m_second) < 0.000001)
     {m_display = "除数不能为零";
      UpdateData(false);
      return;
     }

     m_second = 1.0/m_second;
     UpdateDisplay(m_second);
}


//“.”按钮的消息响应函数
void CMyCalculatorDlg::OnButtonPoint()
{
     m_coff = 0.1 ;

}
//“+/-”按钮的消息响应函数
void CMyCalculatorDlg::OnButtonSign()
```

```
{
    m_second = -m_second;
    UpdateDisplay(m_second);
}
```

**6.** 查看结果

编译、连接运行程序就得到图 4.1 所示的计算器界面。

### 4.2.7　技术要点

**1.** 关于 **UpdateData**()函数

UpdateData 函数是 CDialog 的基类 CWnd 的成员函数，函数原型如下：

```
BOOL UpdateData( BOOL bSaveAndValidate = TRUE );
```

一般在对话框的派生类中，利用 UpdateData()函数进行控件和相应变量之间的数据传递，形式如下：

```
UpdateData(TRUE);  //将控件中的数据传递给相应的变量
UpdateData(FALSE)  //将变量中的数据传递给相应的控件，即显示
```

例如，在 UpdateDisplay()函数中，首先将函数参数 lVal 通过转化和截去零操作，赋值给与编辑框控件相关联的变量 m_display，最后调用 UpdateData(false)使变量 m_display 的值显示编辑框中。

```
void CMyCalculatorDlg::UpdateDisplay(double lVal)
{
    m_display.Format( _T("%f"),lVal);
    int i=m_display.GetLength();
    //格式化输出，将输出结果后的零全部截去
    while(m_display.GetAt(i-1)=='0')
    {m_display.Delete(i-1,1);i--;}
    UpdateData(false); //更新显示编辑框变量 m_display
}
```

**2.** 关于消息响应函数

用 MFC ClassWizard 为对话框中某按钮的 BN_CLICKED 事件添加消息处理函数时，MFC ClassWizard 做了三件事：

(1) 在类的定义 MyCalculatorDlg.h 文件中，添加了消息响应函数的函数原型；

(2) 在类的实现文件 MyCalculatorDlg.cpp 中，添加了函数体；

(3) 在类的实现文件 MyCalculatorDlg.cpp 中，添加了消息映射。

所以，如果想删除通过 ClassWizard 创建的消息响应函数，在“MFC ClassWizard”对话框中，选中要删除的函数，单击“Delete function”按钮将函数删除。但是源文件中相应的函数体并没有删除，需要用手工方法将函数体删除。

**3.** 数据交换和校验

对话框数据交换(Dialog data exchange，简称 DDX)可以方便地实现对话框中控件数值的初始

化和获取用户的数据输入。对话框数据校验(Dialog data validation,简称 DDV)可以对对话框中控件的数据进行校验。具体实现时可以通过 ClassWizard 定义与控件关联的数据成员实现 DDX,通过限定数据范围实现 DDV。

例如,在对话框“IDD _ MYCALCULATOR _ DIALOG”中,通过 ClassWizard 对标识号为“IDC _ DISPLAY”的“Edit Box”控件创建了 m _ display 变量,数据类型为 CString。ClassWizard 自动地在文件 MyCalculatorDlg.cpp 中创建了相应的对话框数据交换代码:

```
void CMyCalculatorDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMyCalculatorDlg)
    DDX_Text(pDX, IDC_DISPLAY, m_display);
    //}}AFX_DATA_MAP
}
```

**4.** 字符串 **CString** 类

CString 类的对象由一个长度可变的字符序列组成,包含很多成员函数用来操作字符串,可以很方便地实现对字符串的各种操作。CString 类中的字符是 TCHAR 类型的。

(1) 构造函数

```
CString();                              //产生一个空的 CString 对象
CString(const CString&stringSrc);       //用另一个 CString 对象的值初始化对象
CString(TCHAR ch, int nRepeat = 1);     //用一个字符重复若干次初始化对象
CString(LPCTSTR lpch, int nLength);     //用一个字符数组的指定长度初始化对象
CString(const unsigned char * psz);     //从一个无符号字符指针初始化对象
CString(LPCWSTR lpsz);                  //从一个 Unicode 字符串初始化对象
CString(LPCSTR lpsz);                   // 从一个 ANSI 字符串初始化对象
```

(2) 常用成员函数

CString 类的常用成员函数参见表 4.4。

表 **4.4** **CString** 类的常用成员函数

| 函数原型 | 说明 |
|---|---|
| int GetLength( ) const | 获取 CString 类对象包含的字符串的长度 |
| BOOL IsEmpty( ) const | 测试 CString 类对象包含的字符串是否为空 |
| void Empty( ) | 使 CString 类对象包含的字符串为空字符串 |
| TCHAR GetAt(int nIndex)const | 获取字符串指定位置处的字符 |
| void SetAt(int nIndex,TCHAR ch) | 设定字符串指定位置处的字符 |
| TCHAR operator[ ](int Index)const | 获取字符串指定位置处的字符 |
| operator LPCTSTR( )const | 返回指向存储在 CString 类对象内的字符串指针 |
| int Compare(LPCTSTR lpsz)const | 比较两个字符串,类似于 C 中 strcmp( )函数 |
| int CompareNoCase(LPCTSTR lpsz)const | 类似于 C 中 Compare( )函数,但忽略字符大小写 |
| MakeUpper | 将字符串中所有的字符全部转化成大写形式 |
| MakeLower | 将字符串中所有的字符全部转化成小写形式 |

例 **1**　连接字符串。

```
CString m_str1 = "下午";
CString m_str2 = "好!";
CString m_str3 = m_str1 + m_str2;
```

执行第三行后，m_str3 的值为“下午好!"。

例 **2**　比较字符串。

```
CString m_str1 = "a";
CString m_str2 = "b";
int  result = m_str1.Compare(m_str2);
if( result == 0)AfxMessageBox("两者相同");
else  if(result > 0) AfxMessageBox("m_str1 大于 m_str2");
else                 AfxMessageBox("m_str1 小于 m_str2");
```

### 4.2.8　优化 MyCalculator 程序

显然，在 CMyCalculatorDlg.cpp 文件中，对应于数字按钮“0”~“9”的消息处理函数代码十分相似，所以上面的处理方法有点烦琐。幸好 Windows 提供了一种简便方法：利用 ON_COMMAND_RANGE 宏，可以处理一系列控件对同一消息处理函数（比如，OnOperandInput）的响应，。但必须按以下方法修改程序：

（1）手工添加函数声明，也就是在 MyCalculatorDlg.h 中“{{AFX_MSG(CMyCalculatorDlg)”与“}}AFX_MSG”之间，添加下面的语句：

```
afx_msg void OnOperandInput(UINT nID);
```

（2）在 MyCalculatorDlg.cpp 文件中“{{AFX_MSG_MAP(CMyCalculatorDlg)”与“}}AFX_MSG_MAP”之间，加入下面语句：

```
ON_COMMAND_RANGE(IDC_BUTTON0,IDC_BUTTON9, OnOperandInput);
```

特别说明：

ON_COMMAND_RANGE()使用一个消息处理函数来处理对某个 ID 范围内所有控件的命令响应，该宏的原型为：

```
ON_COMMAND_RANGE(ID1,ID2, memberFxn);
```

其中 ID1 是控件 ID 的起始值；ID2 是控件 ID 的结束值；memberFxn 为消息响应函数。

当在资源编辑器中创建一个控件时，Visual C++ 会自动为该控件设立一个值用以标识该控件，在 Resource.h 文件中按如下形式定义：

```
#define  IDC_BUTTON0   1000
……
#define  IDC_BUTTON9   1009
```

而 ON_COMMAND_RANGE 宏所处理的正是所有 ID 值位于 1000~1009 之间的控件。

# 4.3 “口令”对话框

多数应用程序利用“口令”对话框来控制用户的使用权限,“口令”对话框是对话框的一个应用实例。为简单起见,下面以第 3 章的程序为例,在其中添加口令,也就是在程序启动时,先要进行密码校验,如图 4.16 所示。

图 4.16 “口令”对话框

下面先学习有关对话框的知识,然后作为对话框的应用,为 MyHello 程序添加“口令”对话框。

## 4.3.1 预备知识

**1.** 模式对话框与非模式对话框

从使用的角度划分,对话框包括模式对话框(Modal dialog box)和非模式对话框(Modaless dialog box)两大类。

所谓模式对话框,是指打开后直至关闭均不可在应用程序其他位置工作的对话框。比较典型的模式对话框有“打开”、“保存”等对话框。

所谓非模式对话框,是指打开仍可切换到应用程序其他位置工作的对话框。比较典型的非模式对话框的一个例子是 Developer Studio 的查找和替换对话框。

**2.** 对话框的设计方法

模式对话框与非模式对话框在设计、调用及工作方式上有所不同,但是基本设计思想是一致的:设计对话框模式、调用对话框、接受数据、关闭对话框。

设计和使用对话框的方法如下:

(1) 创建对话框资源,并添加各种所需的控件;

(2) 创建对话框类(CDialog)的派生类,并将其与已创建的对话框资源相连接;

(3) 在创建的派生类中添加所需的数据成员变量、数据交换函数(DDX)和数据验证函数(DDV);

（4）为对话框中各控件添加所需的消息响应函数，并编写程序代码；

（5）为对话框的调用者添加对话框的调用。

① 调用模式对话框

首先调用对话框类（CDialog）的两个公共构造函数中的一个构造对话框对象，然后调用对话框对象的 DoModal 成员函数加载对话框资源，显示对话框。用户退出对话框后，DoModal 函数返回调用处。必要时可以利用基类 Cdialog 的成员函数，获取相关信息。

② 调用无模式对话框

在对话框类中编写自己的公共构造函数。在调用时通过调用该构造函数构造对话框对象。调用对话框对象的 Create( )成员函数加载对话框资源，然后根据对话框资源是否具有 WS _ VISIBLE 属性决定是否显示对话框。若无该属性，则需使用 ShowWindow( )成员函数显示对话框。Create( )成员函数加载对话框资源后返回者调用处，其后的返回关闭等操作需另作处理。

**3.** 对话框的调用

假设被调用对话框资源的标识号（ID）为：IDD _ PASSWORD _ DIALOG，创建的对话框类名为：CPasswordDialog，则

（1）调用模态对话框，需添加如下代码：

```
CPasswordDialog  m_Dlg;     //声明 CPasswordDialog 类的一个对象 m_Dlg
m_Dlg.DoModal();            //显示模态对话框
```

（2）调用非模态对话框，需做下列工作

① 在视图类的头文件中添加 CPasswordDialog 的指针变量：

```
CPasswordDialog *m_Dlg;
```

② 在视图类的构造函数中将其初始化：

```
m_Dlg = NULL;
```

③ 在调用函数中添加如下代码：

```
if(m_Dlg == NULL)  //如果指针指针向 NULL，说明对话框还没有创建，动态创建
{  m_Dlg = new CPasswordDialog;
   m_Dlg->Create(IDD_PASSWORD_DIALOG,this);
}

m_Dlg->ShowWindow(SW_SHOW);   //显示非模态对话框
```

有了这些预备知识，下面开始为 MyHello 程序添加口令对话框。

### 4.3.2　编辑“口令”对话框资源

**1.** 创建对话框资源

参照第2章中2.3.2节的方法，在 MyHello 工程中，新建一个 ID（标识符）为“IDD _ PASSWORD _ DIALOG”、“Caption”为“口令”的对话框资源。

**2.** 编辑对话框资源

按表 4.5 设计“IDD _ PASSWORD _ DIALO”G 对话框，设计完的对话框如图 4.17 所示。

表 **4.5** “**IDD _ PASSWORD _ DIALOG**”对话框中控件属性表

| 对　象 | 属　性 | 属性值 |
|---|---|---|
| Edit Box | ID<br>Password<br>Border | IDC _ PASSWORD _ EDIT<br>Checked(styles)<br>Checked(styles) |
| Button | ID<br>Caption | IDOK<br>确　认 |
| Button | ID<br>Caption | IDCANCEL<br>取　消 |
| Static Text | ID<br>Caption | IDC _ STATIC<br>请输入口令： |

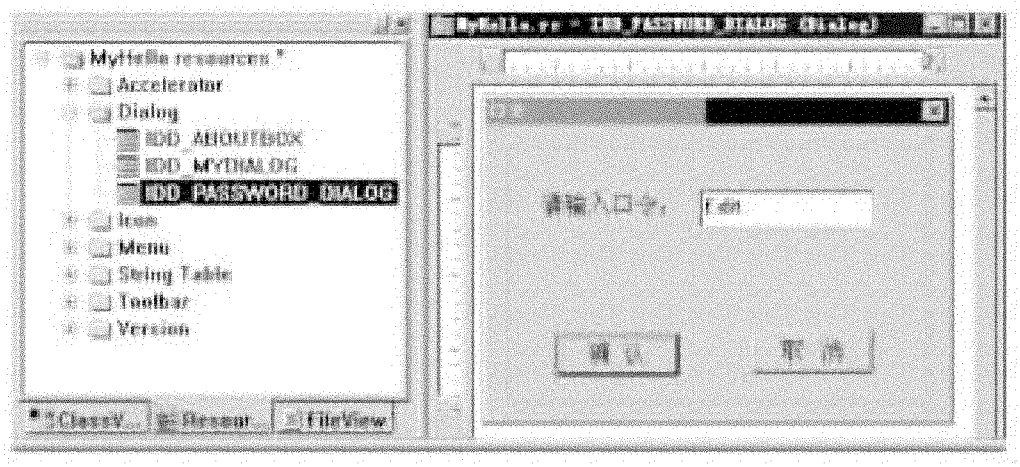



图 4.17　编辑好的“口令”对话框

### 4.3.3　创建“口令”对话框类

为“IDD _ PASSWORD _ DIALOG”对话框创建一个新类。操作步骤如下：

（1）在“IDD _ PASSWORD _ DIALOG”对话框资源上单击鼠标右键，在弹出的快捷菜单中选择“Class Wizard”(或双击对话框，或通过菜单 View/Class Wizard)，弹出“MFC ClassWizard”对话框。在“MFC ClassWizard”对话框弹出后紧接着弹出“Adding Class”对话框。

特别说明：

在打开“MFC ClassWizard”对话框过程中，如果系统检查到有新建对话框资源、菜单资源等，并且该资源没有与相应的类关联，就会弹出“Adding Class”对话框，询问是否创建新类或与某类关联。

（2）在“Adding Class”对话框中确认默认选择“Create a new class”，单击“OK”按钮，弹出“New Class”对话框。

（3）在“Name”编辑框中输入新建类名“CPasswordDialog”，其他取默认值，如图 4.18 所示。

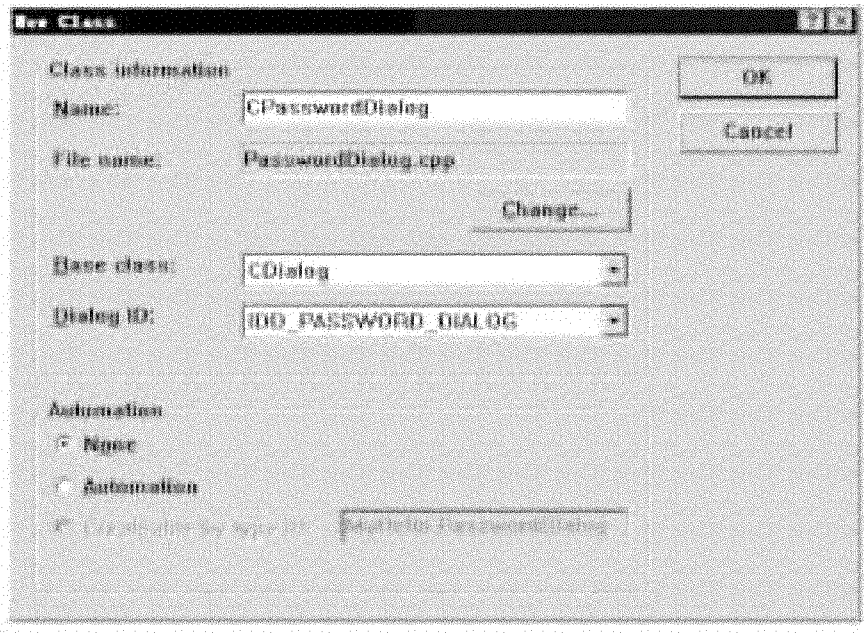



图 4.18　“New Class”对话框

（4）单击“OK”按钮，显示“MFC ClassWizard”对话框。

（5）单击“MFC ClassWizard”对话框中的“OK”按钮，此时，在 MyHello 工程中添加了一个 CpasswordDialog 类，从而也就新创建了 PasswordDialog.h 和 PasswordDialog.cpp 两个文件，如图 4.19 所示。

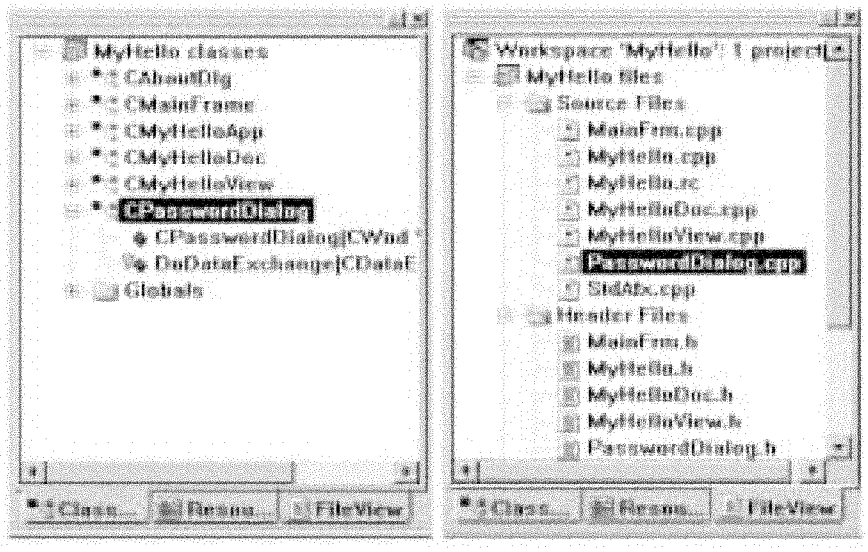



图 4.19　新建了一个类 CpasswordDialog 及类的 2 个构成文件

### 4.3.4　为“口令”编辑框引入变量

为了能够在程序运行过程中访问口令对话框中的编辑框，必须为它引入一个变量，从而能够

使编辑框以变量的形式出现在程序中。

为“IDC _ PASSWORD _ EDIT”编辑框引入变量的步骤如下：

（1）在“View”菜单中选择“ClassWizard”菜单项。

（2）在“MFC ClassWizard”对话框中，选择“Member Variable”标签，进行如下设置：

```
Class name: CPasswordDialog
Control IDs: IDC _ PASSWORD _ EDIT
```

（3）单击“Add Variable...”按钮，此时，Visual C++ 6.0 将显示一个“Add Member Variable”对话框，设置如下：

```
Variable name:   m _ password
Category:        Value
Variable type:   CString
```

（4）单击“Add Member Variable”对话框的“OK”按钮，返回到“MFC ClassWizard”对话框，再单击“MFC ClassWizard”对话框的“OK”按钮。于是，Visual C++ 6.0 就为编辑框“IDC _ PASSWORD _ EDIT”引入变量 m _ password，如图 4.20 所示。

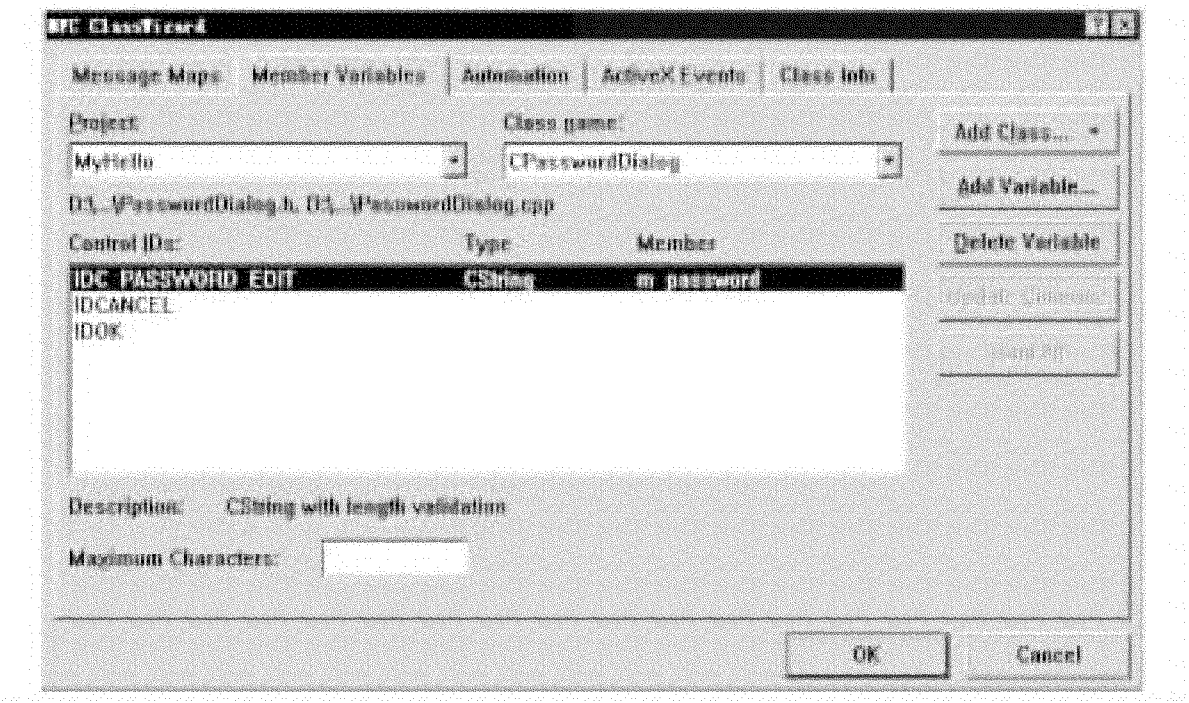



图 4.20 为编辑框“IDC _ PASSWORD _ EDIT”引入变量 m _ password

### 4.3.5 调用“口令”对话框

以模式对话框方式，启动口令对话框。

在 MyHello 程序中，要求经口令验证才能启动程序，必须在程序框架还没有出现时就弹出口令对话框，所以在应用程序类的 InitInstance( )函数中编写调用“口令”对话框代码，进行口令的检查。

（1）修改 InitInstance( )函数。

```
BOOL CMyHelloApp::InitInstance()
{
   AfxEnableControlContainer();
   ……
   m_pMainWnd->ShowWindow(SW_SHOW);
   m_pMainWnd->UpdateWindow();
   CPasswordDialog Dlg; //声明 CpasswordDialog 类的对象
   if(Dlg.DoModal() != IDOK)return false;//用户按下的不是“确认”按钮
   if(Dlg.m_password != "123456")//口令设为字符串"123456"
   {AfxMessageBox("口令错误,确认后将退出程序");
   return false;
   }
   return TRUE;
}
```

（2）在 MyHello.cpp 的头部，加入如下的包含语句：

```
#include"PasswordDialog.h"
```

（3）查看结果。

编译、连接运行程序就得到如图 4.16 所示的口令验证对话框。当输入正确口令：123456 时，单击“确认”按钮，就正常启动 MyHello.exe 程序。倘若输入的口令有误，则程序会提示“口令错误，确认后将退出程序”信息。

### 4.3.6　显示非模式对话框

和模态对话框不同，非模态对话框的生存周期要求比较长，一般和父级窗口的生存周期相同。

例如，在 MyHello 程序中，添加一菜单项，当点击该菜单项时，将“口令”对话框以非模式显示。请读者按以下要求编辑一菜单：

```
ID:ID_NOMODALLESS;
Caption: 非模式对话框。
```

显示非模式对话框操作步骤如下：

（1）在视图类中添加对话框成员指针。

```
class CMyHelloView : public CView
{
protected: // create from serialization only
      CMyHelloView();
      DECLARE_DYNCREATE(CMyHelloView)

// Attributes
public:
      CMyHelloDoc* GetDocument();
      ……
```

```
// Implementation
public:
     CPasswordDialog * m_pModallessDlg;
     virtual ~CMyHelloView();
     ……
};
```

为了使用 CpasswordDialog 类，需要包含该类的头文件。在 MyHelloView.h 中加入：

```
#include"PasswordDialog.h"
```

（2）修改视类的构造函数和析构函数。

分别在视类的构造函数和析构函数中添加如下代码：

```
CMyHelloView::CMyHelloView()
{
   // TODO: add construction code here
   m_nLine = 0;
   m_pModallessDlg = NULL; //初始化指针
}

CMyHelloView::~CMyHelloView()
{
     if(m_pModallessDlg != NULL)
          delete m_pModallessDlg;//释放空间
}
```

（3）编写调用非模式对话框代码。

① 选择“View”菜单中的“ClassWizard”菜单项，弹出“MFC ClassWizard”对话框。

② 在“MFC ClassWizard”对话框中，做如下设置：

```
Object IDs:    ID_NOMODALLESS
Class name:    CMyHelloView
Messages:      COMMAND
```

③ 单击“Add Function…”按钮，弹出“Add Function”对话框，求默认函数名。

④ 单击“OK”按钮，就添加了消息响应函数 OnNomodalless()。

⑤ 单击“Edit Code”按钮，定位在 OnNomodalless()函数，编写如下显示非模式对话框代码：

```
void CMyHelloView::OnNomodalless()
{
     // TODO: Add your command handler code here
     if(m_pModallessDlg == NULL)//如果指针指针向 NULL，说明对话框还没有创建，动态创建
          {    m_pModallessDlg = new CPasswordDialog;
               m_pModallessDlg->Create(IDD_PASSWORD_DIALOG,this);
          }
     m_pModallessDlg->ShowWindow(SW_SHOW);
}
```

（4）查看结果。

编译、连接运行程序后，点击“非模式对话框”菜单项，启动口令对话框口，此时，用户仍可切换到应用程序其他位置工作。

## 4.4　通用对话框

在应用程序设计中，除使用自定义的对话框之外，经常用到系统定义的通用对话框类。通用对话框类用于各种 Windows 常用应用程序中，执行各种标准操作。它们由 CDialog 类派生而来，其资源全部包含在\Windows \SYSTEM 目录下的 COMMDLG.DLL 动态链接库中。表 4.6 列出了通用对话框类的名称及其作用。

表 **4.6**　通用对话框类的名称及其作用

| 类名 | 作用 | 数据成员及其结构 | 常用成员函数 | 作用 |
|---|---|---|---|---|
| CFileDialog | 打开或保存或新建文件 | m _ ofn:<br>OPENFILENAME | GetFileName<br>GetPathName<br>GetFileExt<br>GetFileTitle<br>GetNextPathName<br>GetReadOnlyPref<br>GetStartPosition | 返回所选文件名<br>返回所选文件路经<br>返回所选文件扩展名<br>返回所选文件标题<br>返回下一个所选文件路经<br>返回所选文件的只读属性<br>返回文件列表中第一个元素的位置 |
| CFontDialog | 从提供的字体列表中选择一种字体 | m _ cf:<br>CHOOSEFONT | GetColor<br>GetCurrentFont<br>GetFaceName<br>GetStyleName<br>GetSize<br>IsUnderline<br>IsBold<br>IsItalic | 获得字体颜色<br>获得当前字体<br>获得字体的名称<br>获得字体的风格名称<br>获得字体的点大小<br>字体是否带下划线<br>确定字体是否粗体<br>确定是否是斜体 |
| CColorDialog | 选择或创建颜色 | m _ cc:<br>CHOOSECOLOR | GetSavedCustomColor<br>SetCurrentColor<br>GetColor | 获得用户创建的定制颜色<br>将当前颜色设定为指定颜色<br>返回一个 COLORREF 结构，其中包括所选的颜色 |
| CPrintDialog | 设置打印机及与打印有关的参数，打印文档 | m _ pd:<br>RRWTDLG | GetCopies<br>GetDeviceName<br>GetDevMale<br>GetDriverName<br>GetFromPage<br>GetToPage<br>GetPortName<br>GetPrinterDC<br>GetDefaults | 获得打印份数<br>获得当前所选的打印设备名称<br>获得一个 DEVMODE 结构<br>获得当前所选的打印驱动名称<br>获得打印队列中的开始页<br>获得打印队列中的结束<br>获得当前所选的打印端口名称<br>获得一个打印设备的控制<br>获得未在对话框中显示的设备默认值 |

### 4.4.1 文件对话框类 CFileDialog 的使用方法

(1) 构造 CFileDialog 类的对象。

CFileDialog 的构造函数原型为：

```
CFileDialog(BOOL bOpenFileDialog,LPCTSTR lpszDefExt = NULL,
            LPCTSTR lpszFileName = NULL,
            DWORD dwFlags = OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
            LPCSTR lpszFilter = NULL,
            CWnd * pParentWnd = NULL
            );
```

其中参数：

bOpenFileDialog 为 TRUE 时，指明构造一个打开文件的对话框，否则构造一个保存文件的对话框。

lpszDefExt 指定一个默认的扩展名。

lpszFilter 允许设置过滤器来限制选择某些文件类型，这个字符串用“|”分割，以“||”结束。例如要处理以 dat 和 doc 为后缀的文件，可以按下面所列设置过滤器：

```
CString lpszFilter = "Dat Files(*.dat)|*.dat|Document(*.doc)|*.doc||";
```

(2) 通过对象的数据成员 m_ofn 初始化字体对话框中各控件的值或状态。

(3) 通过成员函数 DoModal()调用对话框。

(4) 应用程序可通过 CFileDialog 类的成员函数获得各种信息。

例如，调用图 4.21 所示的打开文件对话框的关键代码如下：

```
CString strFilter = "Dat Files(*.dat)|*.dat|All Files(*.*)|*.*||";
//打开文件对话框
CFileDialog FileDlg(true, NULL, NULL,
                    OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
                    (LPCSTR)strFilter, this);
if(FileDlg.DoModal() != IDOK)return;
CString strFileName = FileDlg.GetPathName();
//打开文件 strFileName
……
```

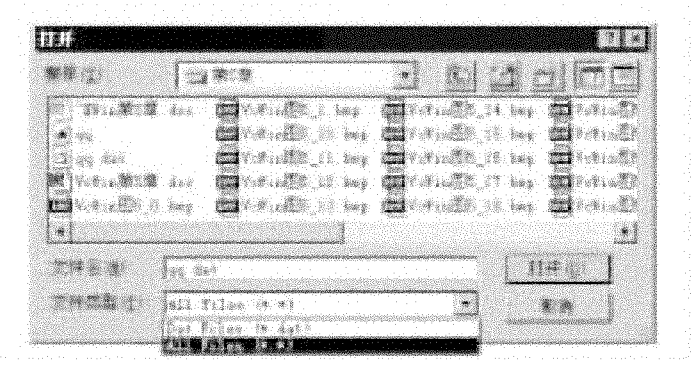


图 4.21 打开文件对话框

### 4.4.2　字体对话框类 CFontDialog 的使用方法

（1）构造 CFontDialog 类的对象。
（2）通过对象的数据成员 m_cf 初始化对话框各控件的值或状态。
（3）通过成员函数 DoModal()调用对话框。
（4）应用程序通过 CFontDialog 类的成员函数获得各种信息。

### 4.4.3　颜色对话框类 CColorDialog 的使用方法

（1）构造 CColorDialog 类的对象。
（2）通过对象的数据成员 m_cc 初始化字体对话框中各控件的值或状态。
（3）通过成员函数 DoModal()调用对话框。
（4）应用程序可通过 CColorDialog 类的成员函数获得各种信息。

### 4.4.4　打印对话框类 CPrintDialog 的使用方法

（1）构造 CPrintDialog 类的对象。
（2）通过对象的数据成员 m_pd 初始化字体对话框中各控件的值或状态。
（3）通过 DoModal()成员函数调用对话框。
（4）应用程序可通过 CPrintDialog 类的成员函数获得各种信息。

## 4.5　常用控件介绍

### 4.5.1　Button 控件

**1. Button**(按钮)控件的消息

控件是为窗口提供信息的部件。Button 控件是对话框中最基本的控件，在对话框中加入的 Button 控件无需任何初始化就可以在对话框的 DoModal 的函数调用中同对话框一起显示出来，只不过不能实现任何功能而已。

当一个按钮被按下时，它将发送一条消息给对话框，对话框类可以建立自己的消息循环并且可为之编写相应的消息响应函数。这是一种最简单的使用 Button 控件的方法。与 Button 控件相关的消息如表 4.7 所示。

表 **4.7**　**Button** 控件的消息

| 消　息 | 含　义 |
| --- | --- |
| BN_CLICKED | 当用户单击按钮时，由按钮发送给对话框 |
| BN_DOUBLECLICKED | 当用户双击按钮时，由按钮发送给对话框 |
| BN_KILLFOUCS | 当按钮失去输入焦点时，由按钮发送给对话框 |
| BN_SETFOUCS | 当按钮获得输入焦点时，由按钮发送给对话框 |

**2. Button** 控件类的成员函数

可用以下语句创建按钮：

```
BOOL Create(LPCTSR lpszCaption, DWORD dwStyle, const RECT &rect,CWnd * pParentWnd,
UINT nID);
```

Button 控件与 CButton 类相关联。可以通过 CButton 类的成员函数对 Button 控件进行设置。CButton 类的成员函数如表 4.8 所示。

表 **4.8** **CButton** 类的成员函数

| 方法原型 | 含义 |
|---|---|
| UINT GetState( ) const | 获取按钮的状态(选中、高亮,得到焦点) |
| void SetState( BOOL bHighlight) | 设定当前按钮状态 |
| Int GetCheck( ) const | 获取按钮控件选中状态 |
| void SetCheck( int nCheck) | 设置按钮的选中状态 |
| UINT GetButtonStyle( ) const | 获取按钮的样式 |
| void SetButtonStyle( Uint nStyle, BOOL bRedraw = TRUE) | 设置按钮的样式 |
| HICON GetIcon( ) const | 获取 SetIcon( )方法设置的图标句柄 |
| HICON SetIcon( HICON hIcon) | 设置按钮上要显示的图标 |
| HBITMAP GetBitmap( ) const | 获取 SetBitmap( )方法设置的位图句柄 |
| HBITMAP SetBitmap( HBITMAP hBitmap); | 设置按钮上要显示的位图 |
| HCURSOR GetCusor( ) | 获取 SetCursor( )方法设置的按钮区域特定光标 |
| HCURSOR SetCursor( HCURSOR hCursor) | 设置按钮区域的光标 |

例如,设与 ID 是 IDC _ BUTTON _ OK 的按钮相关联的对象为:m _ button _ ok,则

```
m_button_ok.EnableWindow(FALSE);
```

使按钮变灰(禁止)。

而语句

```
m_button_ok.SetState(1);
```

使按钮高亮。

## 4.5.2 Edit 控件

Edit 控件是对话框实现输入输出的重要人机交互接口。通过 Edit 控件,用户可以输入文本信息,可以将输入信息转换为各种类型的数据。相反,也可以将某变量的值通过 Edit 控件显示。

Edit 控件除了常见的单行编辑控件外,还有多行 Edit 控件。一个多行 Edit 控件稍加改进就能成为一个功能简单的编辑器。事实上 Windows 中自带的记事本应用程序就是一个带有菜单的多行 Edit 控件。

**1. Edit** 控件的消息

当用户对 Edit 控件的文本进行编辑操作时,Edit 控件可能向对话框发送如表 4.9 所示的消息。

表 4.9　Edit 控件的消息

| 消　息 | 含　义 |
| --- | --- |
| EN _ CHANGE | 当用户改动窗口的文本后，控件向父窗口发出此消息，与 EN _ UPDATE 不同，EN _ CHANGE 是在窗口的文本被刷新显示后，修改已成定局时，Edit 控件发出的消息 |
| EN _ ERRSPACE | 当前的 Edit 控件不能分配足够内存来满足用户的要求时发送的消息 |
| EN _ HSCROLL | 用户操作了 Edit 控件的水平滚动条，在编辑窗口被刷新前，Edit 控件发出的消息 |
| EN _ KILLFOUCS | 当前 Edit 控件失去输入焦点时，Edit 控件向父窗口发出此消息 |
| EN _ MAXTEXT | 当前的输入字符数超过了预先限定的字符数并被截去超出部分时，Edit 控件向窗口发出此消息，或者在 Edit 控件指定为 ES _ AUTOHSCROLL 风格时，输入总行数超过 Edit 控件宽度时，也将发出此消息。此外，当 Edit 控件指定 ES _ AUTOVSCROLL 风格时，输入总行数超过 Edit 控件高度时，控件也将发出此消息 |
| EN _ SETFOCUS | 当 Edit 控件获取到输入焦点时，Edit 控件发出此消息 |
| EN _ UPDATE | 在编辑窗口即将显示更新过的文本前，控件已完成重新格式化文本，Edit 控件发出此消息，此时窗口的大小可以以改变 |
| EN _ VSCROLL | 用户操作过垂直滚动条，在编辑窗口被刷新前，Edit 控件发出的消息 |

Edit 控件与 CEdit 类相关联。可以通过 CEdit 类的成员函数对 Edit 控件进行设置。CEdit 类的成员函数如表 4.10 所示。

表 4.10　CEdit 类的成员函数

| 函数名 | 含　义 |
| --- | --- |
| void SetReadOnly( BOOL h = TRUE) | 设置 Edit 控件为只读状态 |
| BOOL CanUndo( ) | 决定一个编辑操作是否可以撤消 |
| BOOL GetModify( ) | 用来判断 Edit 控件中包含的文字内容是否被修改 |
| void SetModify( BOOL bModified = TRUE) | 设置当前文本的修改标志 |
| GetRect( ) | 获取 Edit 控件文本框的大小 |
| EmptyUndoBuffer( ) | 清除 Edit 控件的撤消操作标志，此后 Edit 控件中不能撤销最后的操作 |
| GetPasswordChar( ) | 在用户输入文本时，获取编辑控件中显示的密码字符 |
| SetPasswordChar( ) | 在用户输入文本时，设置或清除在 Edit 控件中显示的密码字符，此后用户不论输入什么字符都将显示为新的密码字符 |
| GetFirstVisibleLine( ) | 确定 Edit 控件中顶上的可见行 |
| LineLength( ) | 获得指定行的长度 |
| LineScroll( ) | 滚动多行文本编辑控件中的文本 |
| LineFromChar( ) | 多行文本编辑控件中获得序号为某个值的字符所在行号 |
| LimitText( ) | 限定用户可能在编辑控件中输入的长度 |
| ReplaceSel( ) | 用指定文本替代编辑控件中选择的文本 |
| SetSel( ) | 在编辑控件中选择字符范围 |
| GetSel | 获取编辑控件中当前选定文本的开始与结束位置 |

### 4.5.3 Static Text 控件

静态(Static Text)控件一般用来显示文字提示信息,提示信息还可以通过位图、图标等来表示。它们是用来美化界面的,一般不响应消息。

默认时静态控件都使用标识 IDC_STATIC,如果运行时需要处理静态控件,就必须另外指定一个 ID 号。如果想改变静态控件的内容,可以使用 CWnd::SetWindowText()函数来设置文本,使用 SetBitMap()函数来设置位图,使用 SetIcon()函数来设置图标,也可以以参数 SW_HIDE 或 SW_SHOW 来调用 CWnd::ShowWindow()隐藏或显示控件:

```
GetDlgItem(IDC_STATIC)->ShowWindow(SW_HIDE);
GetDlgItem(IDC_STATIC)->ShowWindow(SW_SHOW);
```

## 习 题 四

1. 制作一个简单对话框,它通过菜单命令弹出,当单击菜单命令“弹出对话框”时,应用程序就弹出一个名为“模态对话框”的对话框,请简单描述开发步骤。

2. 在本章的 4.3.4 中,如果为 IDC_PASSWORD_EDIT 编辑框引入变量为整型,即

```
int  m_password;
```

请修改 InitInstance()函数中启动“口令”对话框的程序段代码:

```
……
CPasswordDialog Dlg; //声明 CpasswordDialog 类的对象
if(Dlg.DoModal()!= IDOK)return false;//用户按下的不是“确认”按钮

if(Dlg.m_password != "123456")//口令设为字符串"123456"
{AfxMessageBox("口令错误,确认后将退出程序");
return false;
}
……
```

3. 完善 InitInstance()函数中启动口令对话框的程序段代码,允许用户最多输入 3 次口令。

4. 在程序中,利用函数 Create()动态创建 Button 控件。

# 第5章

# 文档与视图结构

## 本章导读

MFC 的文档/视图结构将数据的观察与数据的操纵分离，文档类对象通常负责数据的存储、打开和保存等，而视图类对象为用户对数据的浏览、编辑提供适当的方法。

文档/视图结构是 Visual C++ 中一个非常重要的概念。本章将通过开发一个简单的学生档案管理程序，使读者掌握以下内容：

- 对文档/视图结构进行深入理解
- 在视图中显示数据
- 利用 CFile 类对文档数据的存储和装入
- 利用串行化对文档数据的存储和装入

## 5.1 学生档案管理程序

在开始编写学生档案管理程序 MySdi 之前，首先观察一下它的外观和功能：

（1）例程 MySdi.exe 运行窗口如图 5.1 所示，是一个单文档应用程序。

（2）程序主窗口包含 4 个单行编辑框和 1 个多行编辑框，提供数据的录入，一个列表框，用于选择学生学号。

（3）工具栏上新创建两个按钮：“打开”和“另存为”，分别用于文档数据的存储和装入。单击“另存为”按钮，会弹出一个“另存为”文件对话框，输入要存储的文件名后，程序中的学生档案将存储在文件中。

（4）单击“打开”按钮，会弹出一个“打开”文件对话框，选择要打开的文件后，文件中的学

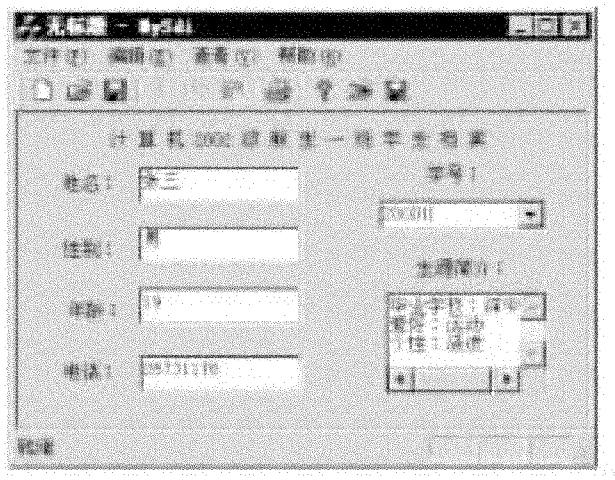


图 5.1 学生档案管理系统

生档案记录将显示在对应的编辑框中。

(5) 在下拉式列表框中,选择某一学号,该学生的相关数据将显示在各编辑框中。

(6) 选择“文件”菜单下的“打开”(或“保存”)子菜单项,程序提供常规的文件打开(或另存为)功能(或单击工具栏上对应按钮)。

按知识点和程序功能,开发工作可分解为:

① 创建一个单文档学生档案管理应用程序,具有录入、浏览功能。

② 改进 MySdi 程序,利用 CFile 类,添加文档数据存储和装入功能。

③ 改进 MySdi 程序,利用 CArchive 类,添加串行化存储和装入功能。

下面,首先对文档/视图的概念稍做介绍,然后分 3 步,完成该系统的开发。

## 5.2 文档与视图的概念

前面一直使用视图与文档,但并没有深入了解它们之间的关系,在这里将开始这个话题。将介绍在程序执行过程中,视图与文档是如何交互的,文档又是如何保存数据的。

### 5.2.1 文档

文档类继承于类 CDocument,它描述了应用的数据。

抽象地说,文档是一个应用程序数据基本元素的集合,它构成应用程序所使用的数据单元,此外文档负责管理和维护应用的数据。

具体一点来说,文档是一种数据源,数据源有很多种,最常见的是磁盘文件,但是文档不必非要是一个磁盘文件,文档的数据源也可以来自串行口或并行口的输入数据。文档对象负责管理来自所有数据源的数据。

### 5.2.2 视图

视图类则继承于视类 CView,它是一个基于视类的窗口。

视图是数据的用户窗口,为用户提供了文档的可视的数据显示,它把文档的部分或全部内容在窗口中显示出来,视图给用户提供了一个同文件中的数据进行交互的界面,它把用户的输入转化为对文档中数据的操作。

### 5.2.3 文档与视图的关系

每个文档都会有一个或多个视图显示,一个文档可以有多个不同的视图。比如,可以将一个集合关系以饼状图的形式显示,也可以将它以数据的形式来显示。总之,要把握一点:文档用来保存数据,视图用来显示数据,视图是显示出的文档。对于文档与视图的关系,可用图 5.2 来描述。

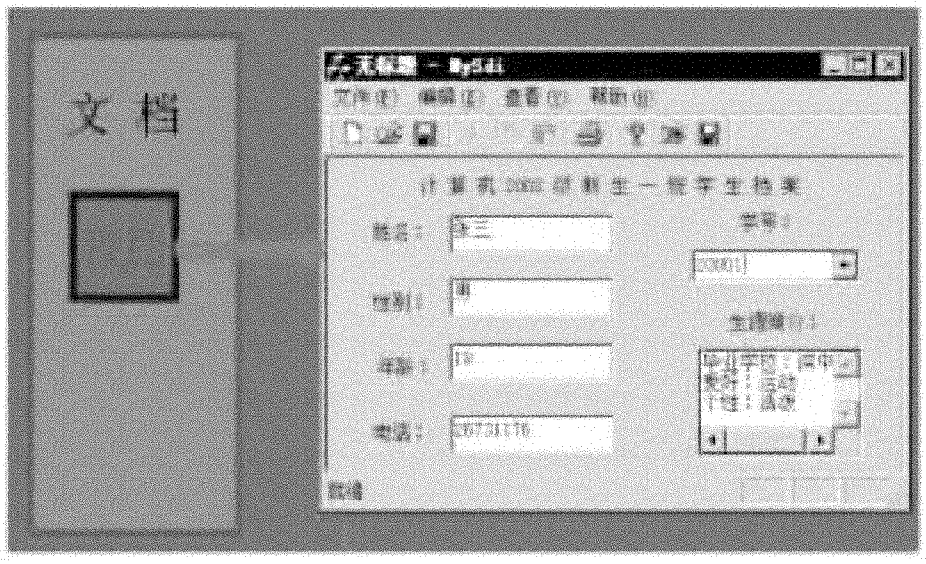



图 5.2　文档与视图关系图

### 5.2.4　文档与视图的交互过程

了解了视图与文档的基本关系后，下面来看文档与视图之间比较复杂的交互过程是如何进行的。简单的说，交互过程中是以下 4 个函数起了关键性的作用。

**1. CView 类的 GetDocument()函数**

一个视图对象只能与一个文档对象相联系。视图类 CView 包含的 GetDocument()函数使用户可以在视图类中得到与当前视图相联系的文档。该函数返回的是一个 CDocument 类或其派生类的指针。从而可以利用得到的文档指针来访问文档中的数据。

例如，前面在视图类中得到了用户输入的数据，此时，视图对象知道当前的数据有了变化，它就应该通知当前的文档，让文档能够及时反映数据的最新的变化，观察前面的例子，在视图类中总能发现类似下面的语句：

```
CMyHelloDoc * pDoc = GetDocument();
ASSERT_VALID(pDoc);
```

第一行便是获得了文档类的指针并将它强制转化为派生类文档的类型。在应用向导生成视类 CView 的派生类时，同时也会创建类似下面所表示的函数：

```
CMySdiDoc * CMySdiView::GetDocument() // non-debug version is inline
{
   ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CMySdiDoc)));
   return (CMySdiDoc *)m_pDocument;
}
```

编译视类对 GetDocument()函数的调用时，编译器调用的实际就是派生视类中的 GetDocument()函数，而不是基类 CView 的函数，所以可以不必用强制类型转换。

**2. CDocument 类的 UpdateAllViews()函数**

上面说的是当视图收到用户数据变化的消息后，会通知文档当前的变化。同样，当文档的数据发生了变化后，文档也要通知视图当前的变化，以便让视图能够及时更新，忠实反映文档的数据。这样的一件工作是通过文档类中的 UpdateAllViews()函数来实现的。

该函数被调用时，应用框架会相应地调用 OnUpdate()函数(下一小节将要论及的函数)。

UpdateAllViews()函数的调用方式会稍微多一些，它可以在文档基类中调用，也可以在派生类中调用，甚至可以在视图类中调用，该函数的原型为：

```
void UpdateAllViews(
            CView * pSender,
            LPARAM lHint = 0L,
            CObject * pHint = NULL
            );
```

在派生文档类中调用该函数时，它的第一个参数 pSender 应该是 NULL；若以下面的形式调用该函数，则会通知所有的视图进行更新。

```
UpdateAllViews(NULL);
```

如果是在视类中调用，则应该有下面的形式：

```
GetDocument()->UpdateAllViews(this);
```

第二个参数 LPARAM lHint 含有与修改有关的信息，第三个参数 CObject * pHint 是一个指向保存修改信息对象的指针，它被声明为是 CObject 类型的，由于 CObject 类是绝大部分 MFC 类的基类或父类，故它可以传递几乎所有 MFC 类的指针。

**3. CView 类的 OnUpdate()函数**

当文档类调用 UpdateAllViews()函数来通知更新视图时，OnUpdate()函数便会被调用。该函数是一个虚函数，其原型如下：

```
virtual  void OnUpdate(
         CView * pSender,
         LPARAM  lHint,
         CObject * pHint
         );
```

各个参数的含义与 UpdateAllViews()函数的参数相同。

当该函数被调用时，它会对文档类进行访问，读取文档的数据，然后对视图进行刷新。它的原理是该过程使视图的某一个部分无效，触发了对视图类成员函数 OnDraw()的调用，从而重新绘制视图客户区。默认情况下 OnUpdate()函数是使整个客户区都无效。用户可以重载该函数以使视图能够反映文档的最新更新情况。

**4. CView 类的 OnInitialUpdate()函数**

该函数原型为：

```
virtual void OnInitialUpdate();
```

当应用程序被启动或者用户选择了打开文件或新建文件时，OnInitialUpdate()函数都会被调用。CView类中的该函数只是调用了OnUpdate()函数，并没有做其他方面的事，用户要对派生类的OnInitialUpdate()函数进行初始化，则可以调用基类的该函数，也可以直接调用派生类的OnUpdate()函数。

# 5.3　单文档应用程序(SDI)

下面开始编写一个具有录入、浏览功能的学生档案管理应用程序。

## 5.3.1　创建工程

创建MySdi工程的步骤如下：

(1) 启动Visual C++6.0。从“File”菜单中选择“New”。此时，显示一个“New”对话框。

(2) 在“New”对话框中选择“Projects”标签 。然后选择“MFC AppWizard(exe)”类型，在“Project name”文本编辑框中输入“MySdi”，单击位于“Location”框右边的小按钮，再从下拉的对话框中选择“d:\MYVC”目录，使新创建的工程文件放置在“d:\MYVC”目录之下，如图5.3所示。

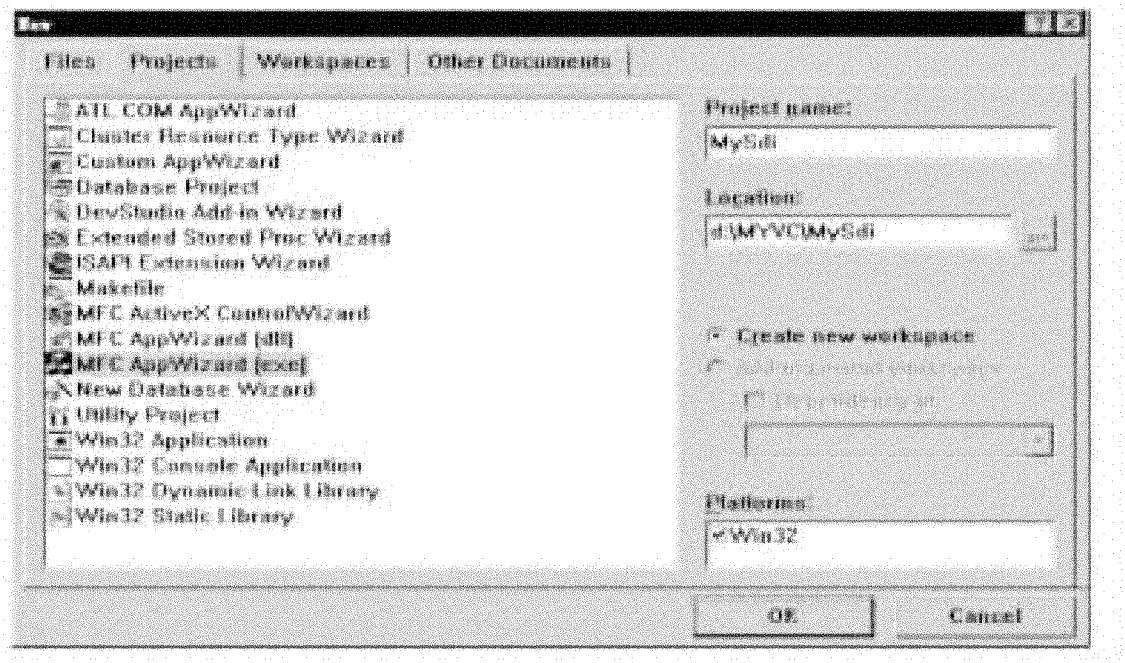



图5.3　指定工程类型、工程名字和工程位置

(3) 单击“New”对话框的“OK”按钮。此时Visual C++6.0将显示“MFC AppWizard - Step 1”对话框。在“MFC AppWizard - Step 1”对话框中：

- Single document选项表示单文档界面，简称SDI，这种类型应用程的主窗口只能容纳一个文档，如Windows自带的记事本。
- “Multiple documents”选项表示多文档界面，简称MDI，这种类型应用程序允许同时打开多

个文档,这些文档可以层叠于主窗口。Microsoft Office 产品就属于 MDI 应用程序。

• “Dialog based”选项表示生成基于对话框的应用程序。

在本例中选择“Single document”,创建一个基于单文档界面的应用程序。然后选择资源语言,本例中选择“简体中文”,如图 5.4 所示。

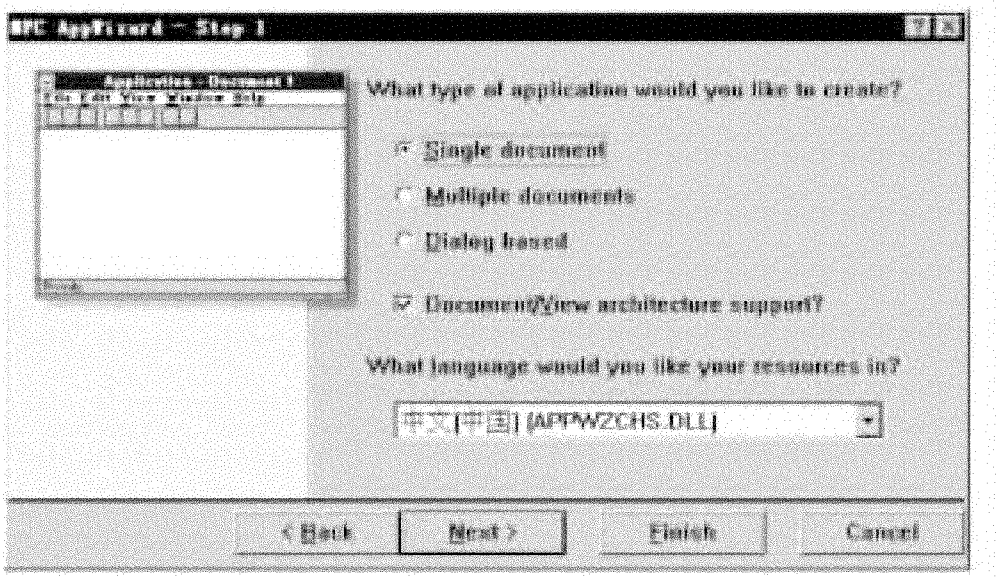



图 5.4 “MFC AppWizard - Step 1”:设置应用程类型

(4) 单击“Next”按钮,在 Visual C++6.0 显示的“MFC AppWizard - Step 2 of 6”对话框中,接受默认的“None”选项。

(5) 单击“Next”按钮,在 VC++6.0 显示的“MFC AppWizard - Step 3 of 6”对话框中,接受默认设置。

(6) 单击“Next”按钮,在 Visual C++6.0 显示“MFC AppWizard - Step 4 of 6”对话框中,接受默认设置。

(7) 单击“Next”按钮,在 Visual C++6.0 显示为“MFC AppWizard - Step 5 of 6”对话框中,接受默认设置。

(8) 单击“Next”按钮,Visual C++6.0 显示为“MFC AppWizard - Step 6 of 6”对话框中,如图 5.5 所示。其中显示“MFC AppWizard”为应用程序创建的所有类以及各个类对应的基类和相应的文件,选取 CFormView 作为视图基类(可以在主窗口内添加控件),其余接受默认设置。

如想修改前面某步的设置,请单击“Back”按钮,可重新设置。

(9) 单击“Finish”按钮,结束“MFC AppWizard”的设计工作,此时 Visual C++6.0 将显示“NewProject Information”窗口,其中显示前 6 步所做的选择的汇总信息。

(10) 单击“OK”。于是,Visual C++6.0 就会创建 MySdi 工程以及相关的所有文件。

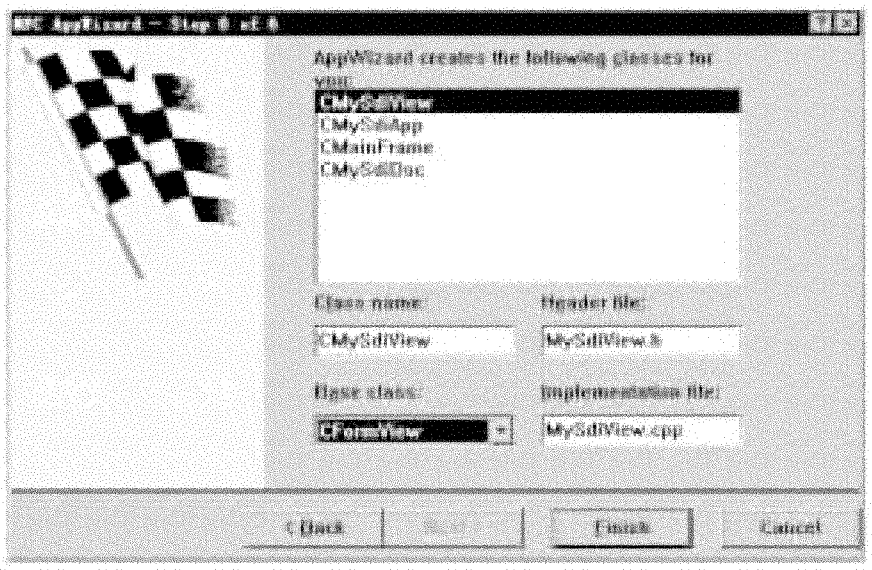



图5.5　“MFC AppWizard – Step 6 of 6”对话框：设置基类

### 5.3.2　可视化设计

设计主窗口的步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 MySdi resources 扩展开，然后再扩展 Dialog，最后，双击“IDD _ MYSDI _ FORM”项，Visual C++ 6.0 显示出处于设计状态的“IDD _ MYSDI _ FORM”对话框。

（2）从“IDD _ MYSDI _ FORM”对话框中删除 TODO 文本。

（3）将鼠标放置工具栏的任一位置，单击鼠标右键弹出工具箱对话框，选择“Controls”复选项（如图 5.6(a)所示），将弹出控件工具箱，如图 5.6(b)所示。

（4）利用工具箱提供的控件，根据表 5.1中的定义编辑对话框资源，设计的对话框如图 5.7 所示。

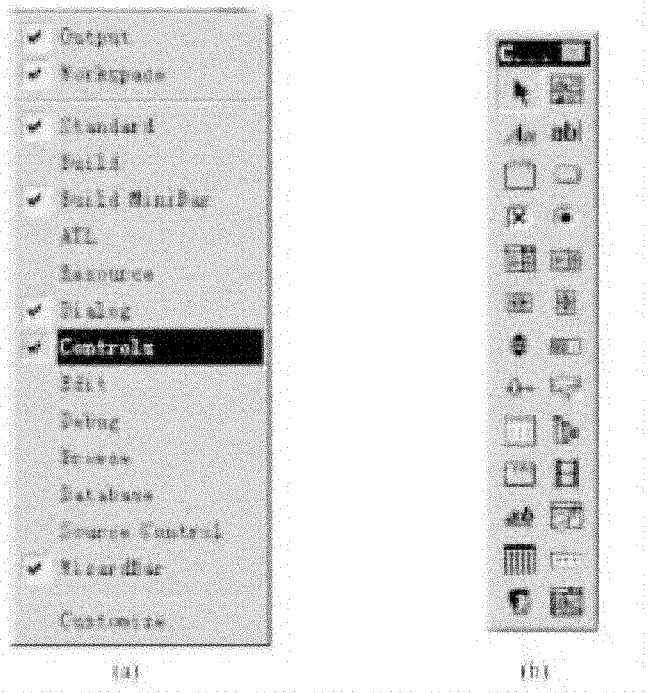



图5.6　工具栏与工具箱

表 5.1 对话框“IDD _ MYSDI _ FORM”中各控件的属性表

| 对象 | 属性 | 属性值 | 对象 | 属性 | 属性值 |
|---|---|---|---|---|---|
| Static Text | ID<br>Caption | IDC _ STATIC<br>姓名: | Edit Box | ID<br>Caption | IDC _ EDIT _ NAME |
| Static Text | ID<br>Caption | IDC _ STATIC<br>性别: | Edit Box | ID<br>Caption | IDC _ EDIT _ SEX |
| Static Text | ID<br>Caption | IDC _ STATIC<br>年龄: | Edit Box | ID<br>Caption | IDC _ EDIT _ AGE |
| Static Text | ID<br>Caption | IDC _ STATIC<br>电话: | Edit Box | ID<br>Caption | IDC _ EDIT _ TEL |
| Static Text | ID<br>Caption | IDC _ STATIC<br>学号: | Combo Box | ID<br>Type<br>Sort,<br>VerticalScroll | IDC _ COMBO<br>Dropdown<br>Checked(styles)<br>Checked(styles) |
| Static Text | ID<br>Caption | IDC _ STATIC<br>生源简介: | Edit Box | ID<br>Multiline<br>Horizontal scroll<br>Vertical scroll | IDC _ EDIT _ SCHOOL<br>Checked<br>Checked<br>Checked |

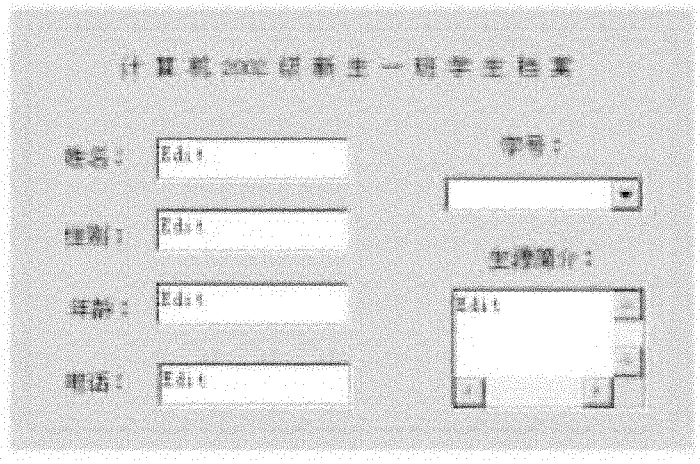


图 5.7 设计完毕的“IDD _ MYSDI _ FORM”对话框

### 5.3.3 给文档类添加成员变量

正如前面所说,文档类是用于存储文档的,它们可以将程序的数据存储到某个文件中,也可以从先前存储的文件中将数据装入。换句话说,文挡类中的数据就是要操作的对象,所以必须将其定义为文档类的数据成员。

**1.** 修改 **CMySdiDoc** 类的定义

在 CMySdiDoc 类中定义结构数组:

```
class CMySdiDoc : public CDocument
```

```
{
protected: // create from serialization only
     CMySdiDoc();
     DECLARE_DYNCREATE(CMySdiDoc)

// Attributes
public:
     struct {
          char name[10];
          char sex[3];
          int  age;
          char tel[14];
          char school[30];
          }m_student[40];
// Operations
……

    DECLARE_MESSAGE_MAP()
};
```

**2.** 初始化变量

在构造函数中对这些变量进行初始化：

```
CMySdiDoc::CMySdiDoc()
{
   // TODO: add one-time construction code here
   for(int i=0;i<40;i++)
   {
   m_student[i].name[0]=NULL;
   m_student[i].sex[0]=NULL;
   m_student[i].age=0;
   m_student[i].tel[0]=NULL;
   m_student[i].school[0]=NULL;
   }
}
```

### 5.3.4 给视图类添加成员变量

视图类是负责屏幕的显示内容。在此，要加入的视图类数据成员实际上就是文档类的数据成员在屏幕上的映射。而将视图类中的变量显示在屏幕上最简捷的方法是：将变量与某编辑控件关联，即为编辑框引入变量。

**1.** 为“**IDD_MYSDI_FORM**”对话框中各控件引入变量

利用 MFC ClassWizard，按表 5.2 所示，为对话框中各控件引入变量，如图 5.8 所示。

表 5.2 用 ClassWizard 增加的关联变量

| 控件 ID | 类型 | 关联变量 | 说明 |
|---|---|---|---|
| IDC _ COMBO | CComboBox | m _ noList | 选择学号的组合框 |
| IDC _ AGE | int | m _ age | 显示年龄编辑框 |
| IDC _ EDIT _ NAME | CString | M _ name | 显示姓名编辑框 |
| IDC _ EDIT _ SCHOOL | CString | m _ school | 显示原毕学校编辑框 |
| IDC _ EDIT _ SEX | CString | m _ sex | 显示性别编辑框 |
| IDC _ EDIT _ TEL | CString | m _ tel | 显示电话号码编辑框 |

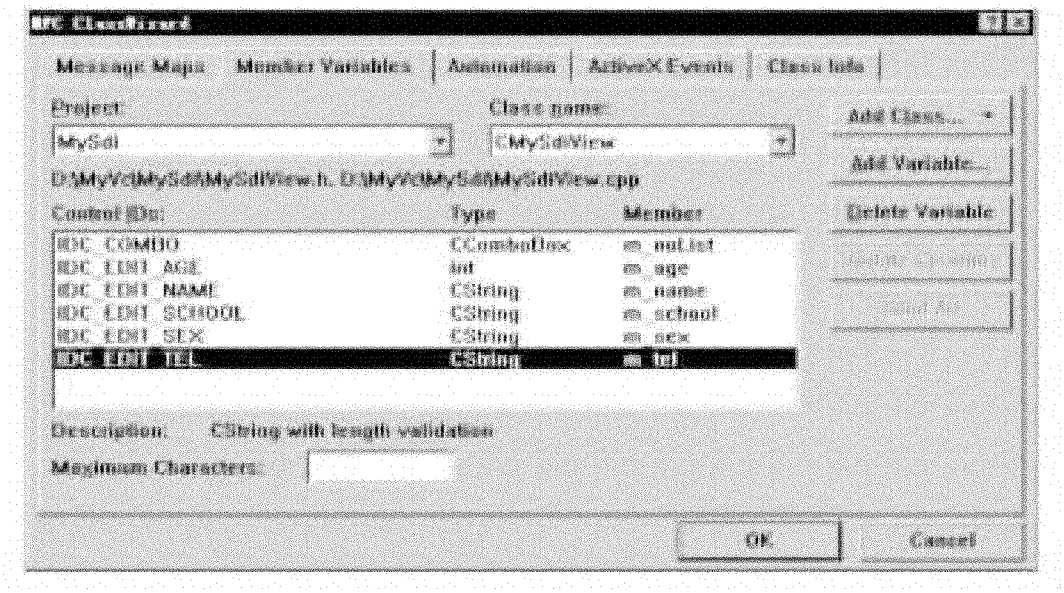



图 5.8 为对话框中各控件引入变量

**2.** 添加一个记录学号的变量

```
class CMySdiView : public CFormView
{
   int  m_nCurrentNo;
protected: // create from serialization only
   CMySdiView();
   DECLARE_DYNCREATE(CMySdiView)
.....
};
```

## 5.3.5 变量初始化

**1.** 修改 **OnInitialUpdate**()函数

修改视类的 OnInitialUpdate()函数:

(1) 设置 Combo Box“学号”。

（2）将文档类中变量赋给对应的编辑控件变量。

```
void CMySdiView::OnInitialUpdate()
{
   CFormView::OnInitialUpdate();
   GetParentFrame()->RecalcLayout();
   ResizeParentToFit();
   char temp[10];
   CString k;
   for(int i=0;i<40;i++)
   {    if(i<10)k="2000";
        else    k="200";
        m_noList.InsertString(i,k+itoa(i,temp,10));
   }

   CMySdiDoc *pDoc = GetDocument();
   m_noList.SetCurSel(0);
   m_name = pDoc->m_student[0].name;
   m_age = pDoc->m_student[0].age;
   m_school=pDoc->m_student[0].school;
   m_sex =pDoc->m_student[0].sex;
   m_tel =pDoc->m_student[0].tel;

   UpdateData(false);
}
```

**2.** 在构造函数中，对变量 **m_nCurrentNo** 初始化

```
CMySdiView::CMySdiView()
     : CFormView(CMySdiView::IDD)
{
     //{{AFX_DATA_INIT(CMySdiView)
     m_name = _T("");
     m_age = 0;
     m_school = _T("");
     m_sex = _T("");
     m_tel = _T("");
     //}}AFX_DATA_INIT
     // TODO: add construction code here
     m_nCurrentNo =0;
}
```

### 5.3.6　处理数据记录的录入

实现数据记录的编辑，就是处理编辑框的 EN_CHANGE 事件。为此，要为编辑框的 EN_CHANGE 事件添加响应函数且要在消息响应函数中，编写将编辑的数据传送给文档类对应的变量的代码。

**1.** 添加消息响应函数

用 MFC ClassWizard 类向导,按照表 5.3 所列添加消息响应函数。

表 **5.3** **ClassWizard** 增加的消息响应函数

| 控件 ID | 事件(消息) | 函数 | 说明 |
|---|---|---|---|
| IDC _ COMBO | CBN _ SELCHANGE | OnSelchangeCombo | 选择学号的组合框 |
| IDC _ EDIT _ NAME | EN _ CHANGE | OnChangeEditName | 编辑与显示姓名 |
| IDC _ AGE | EN _ CHANGE | OnChangeEditAge | 编辑与显示年龄 |
| IDC _ EDIT _ SCHOOL | EN _ CHANGE | OnChangeEditSchool | 编辑与显示原毕业学校 |
| IDC _ EDIT _ SEX | EN _ CHANGE | OnChangeEditSex | 编辑与显示性别 |
| IDC _ EDIT _ TEL | EN _ CHANGE | OnChangeEditTel | 编辑与显示电话号码 |

下面以“姓名”编辑框的 EN _ CHANGE 事件添加消息响应函数为例,说明其操作步骤,其余类似,留给读者练习。

(1) 选择“View”菜单中的“ClassWizard”菜单项。

(2) 在“MFC ClassWizard”对话框中,选择“Message Maps”标签,如图 5.9 所示,进行如下设置:

```
Class name:   CMySdiView
Object IDs:   IDC _ EDIT _ NAME
Message:      EN _ CHANGE
```

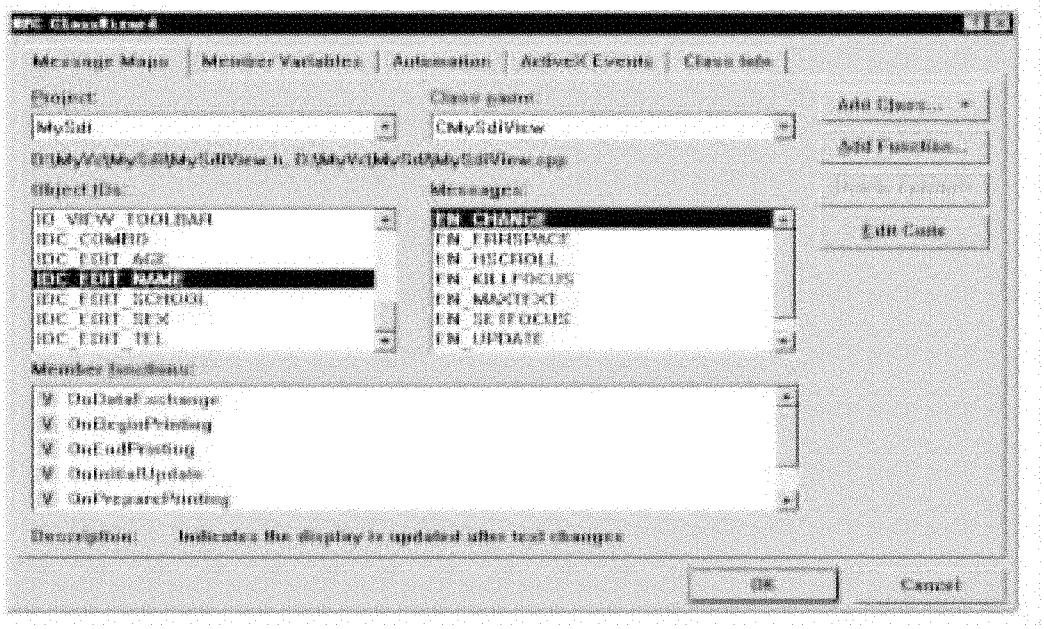



图 5.9 选中“IDC _ EDIT _ NAME”编辑框的 EN _ CHANGE 事件

(3) 单击“Add Function...”按钮,将弹出“Add Member Function”对话框,接受默认函数名,如图 5.10 所示。

(4) 单击“OK”按钮,就为“IDC _ EDIT _ NAME”编辑框的 EN _ CHANGE 事件添加名为 OnChangeEditName 的消息响应函数,如图 5.11 所示。

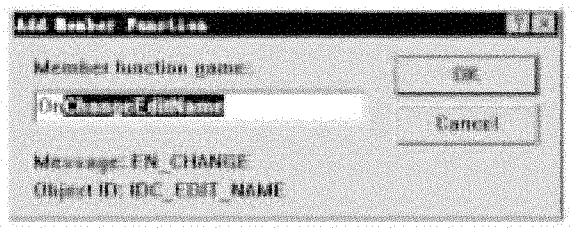



图 5.10　“Add Member Function”对话框

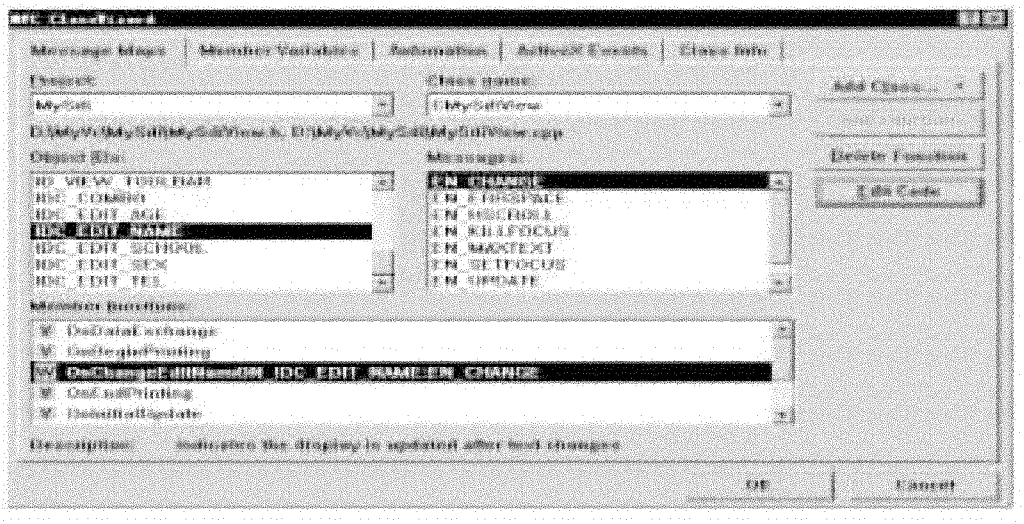



图 5.11　添加 OnChangeEditName 消息响应函数

**2.** 为 **OnChangeEditName**( )函数编写程序代码

在图 5.11 中，点击“Edit Code”按钮(或在 MySdiView.cpp 定位 OnChangeEditName( ))，将定位 OnChangeEditName( )函数，编写如下函数代码，其功能是将视图的数据传送给文档类变量。

```
void CMySdiView::OnChangeEditName()
{
    CMySdiDoc * pDoc = GetDocument();
    UpdateData(true);
    if(strcmp( pDoc->m_student[m_nCurrentNo].name,m_name))
    {strcpy( pDoc->m_student[m_nCurrentNo].name,m_name);
     pDoc->SetModifiedFlag();
    }
}
```

**3.** 代码分析

在 OnChangeEditName( )函数中，语句：

```
CMySdiDoc * pDoc = GetDocument();
```

获取指向文档类对象的指针，用于操作文档类中的数据。语句：

```
UpdateData(true);
```

用控件的值去更新与之关联的变量,从而使 m_name 的值与其关联的"姓名"编辑框一致。最后条件语句:

```
if(strcmp( pDoc->m_student[m_nCurrentNo].name,m_name))
    {strcpy( pDoc->m_student[m_nCurrentNo].name,m_name);
     pDoc->SetModifiedFlag();//设置数据修改标志
    }
```

就是当对应的变量值有所改变时,进行值的传递,这样保证文档类与视图类相对应的变量值时刻保持一致。同时设置数据修改标志,当用户退出程序时,程序会自动提示是否将数据保存。

**4.** 为其他消息响应函数编写程序代码

```
void CMySdiView::OnChangeEditAge()
{
    CMySdiDoc *pDoc = GetDocument();
    UpdateData(true);
    if( pDoc->m_student[m_nCurrentNo].age != m_age)
    {pDoc->m_student[m_nCurrentNo].age = m_age;
     pDoc->SetModifiedFlag();
    }
}

void CMySdiView::OnChangeEditSchool()
{
CMySdiDoc *pDoc = GetDocument();
    UpdateData(true);
    if(strcmp( pDoc->m_student[m_nCurrentNo].school,m_school))
    {strcpy( pDoc->m_student[m_nCurrentNo].school,m_school);
     pDoc->SetModifiedFlag();
    }// TODO: Add your control notification handler code here
}

void CMySdiView::OnChangeEditSex()
{
    CMySdiDoc *pDoc = GetDocument();
    UpdateData(true);
    if(strcmp( pDoc->m_student[m_nCurrentNo].sex,m_sex))
    {strcpy( pDoc->m_student[m_nCurrentNo].sex,m_sex);
     pDoc->SetModifiedFlag();
    }
}

void CMySdiView::OnChangeEditTel()
{
    CMySdiDoc *pDoc = GetDocument();
    UpdateData(true);
    if(strcmp( pDoc->m_student[m_nCurrentNo].tel,m_tel))
    {strcpy( pDoc->m_student[m_nCurrentNo].tel,m_tel);
     pDoc->SetModifiedFlag();
```

```
    }
}
void CMySdiView::OnSelchangeCombo()
{
    // TODO: Add your control notification handler code here
    CMySdiDoc *pDoc = GetDocument();
    m_nCurrentNo = m_noList.GetCurSel();
    m_name = pDoc->m_student[m_nCurrentNo].name;
    m_age = pDoc->m_student[m_nCurrentNo].age;
    m_school = pDoc->m_student[m_nCurrentNo].school;
    m_sex = pDoc->m_student[m_nCurrentNo].sex;
    m_tel = pDoc->m_student[m_nCurrentNo].tel;
    UpdateData(false);
}
```

### 5.3.7　查看结果

代码加上之后，就可以编译、连接运行 MySdi 程序了，操作步骤如下：

（1）选择“Build”菜单中的“Build MySdi.exe”菜单项，Visual C++ 6.0 就会编译并连接 MySdi.exe 程序。

（2）选择“Build”菜单中的“Execute MySdi.exe”菜单项，Visual C++ 6.0 就会执行 MySdi.exe 程序，MySdi.exe 程序主窗口也随之出现。

（3）选取学号，并在各编辑框中输入数据，再次选择已编辑数据的学号时，对应的编辑框中将显示相应的数据。即程序具有数据录入和浏览功能。

（4）单击 MySdi.exe 程序主窗口右上角关闭按钮，退出 MySdi.exe 程序。

### 5.3.8　组合框介绍

在 MySdi 应用程序中，使用了组合框来存储与显示学号。

组合框可以看作是一个编辑框或静态文本框与一个列表框的组合，组合框的名称也正是由此而来的。当前选定的项将显示在组合框的编辑框或静态文本框中，如果组合框具有下拉列表（drop-down list）样式，则用户可以在编辑框架中键入列表框中某一项的首字母，在列表框可见时，与该字母相匹配的最近的项将被加亮显示，在绘制组合框时，可以使用控件的“Properties”对话框设置控件的各种属性式样。表 5.4 给出组合框的三种式样。

表 5.4　Combo Box 控件三种式样

| 选　项 | 风　格 |
| --- | --- |
| Simple | 创建包括编辑框控件和列表框架的简单组合框，其中编辑框控件用来接受用户的输入 |
| Dropdown | 创建下拉组合框，该类型与简单组合框类似，但仅当用户单击编辑框控件部分右边的下拉箭头时组合框的列表框部分才被显示 |
| Drop List | 该类型类似于下拉样式，只是使用静态文本项代替编辑框控件来显示列表框中的当前选择 |

Combo Box 控件与 CComboBox 类关联。CComboBox 类常见的成员函数如表 5.5 所示。

表 5.5 Combo Box 控件的常见的成员函数

| 函 数 | 说 明 |
|---|---|
| GetCount | 获得组合框中列表框项的数目 |
| GetCurSel | 返回组合框中列表框的当前选定项的索引 |
| SetCurSel | 设置组合框中列表框的一个字符串 |
| DeleteString | 从组合框的列表框中删除一个字符串 |
| InsertString | 向组合框的列表框中插入一个字符串 |
| AddString | 向组合框的列表框中添加一个字符串 |
| SelectString | 在组合框的列表框中查找字符串，如果找到，在列表框中选择该字符串，并复制到编辑框控件中 |

# 5.4 文档的存储和装入

几乎每一个应用程序都要进行数据处理，并将处理的数据作为文件保存起来。MFC 提供了大量的函数和类用于文件的读写、遍历、删除和改名等操作。

本节先介绍使用 CFile 类进行文件读写操作，然后改进 MySdi 应用程序。

## 5.4.1 利用 CFile 类操作文件

CFile 是 MFC 文件类的基类，它直接提供二进制文件的输入/输出操作，并通过派生类支持文本文件和内存文件。该类与其派生类的层次关系让程序通过各种 CFile 接口使各种文件操作一致化。不论是内存文件、磁盘文件、二进制文件，还是文本文件都可以使用几乎相同的函数来操作，这使程序设计比较简单。CFile 类的主要成员函数如表 5.6 所示。

表 5.6 CFile 类的主要成员函数

| 成员函数 | 功能 |
|---|---|
| Open | 打开磁盘文件 |
| Read | 从打开的文件中读取数据 |
| Write | 将数据存储在打开的文件中 |
| Close | 关闭打开的文件 |
| Seek | 文件指针定位 |
| Remove | 删除指定的文件 |
| Rename | 更改指定文件名 |
| GetStatus | 获取文件状态信息 |
| SetStatus | 设置文件状态信息 |
| GetLength | 获取文件长度信息 |

利用 CFile 类操作文件的步骤如下：

（1）创建 CFile 类对象或派生类对象。

（2）打开文件。

（3）读/写文件。

（4）关闭文件。

**1.** 文件打开函数 **Open**

CFile 类一般通过成员函数 Open()来打开文件。该函数的原型为：

```
BOOL Open(LPCTSTR lpzsFileName,UINT nOpenFlag,CFileException * pError = NULL);
```

其中参数：

lpzsFileName：为欲打开的文件名，可包含路经名。

nOpenFlag：用于设置访问模式，访问模式的种类，如表 5.7 所示。

表 **5.7**　访问模式

| 访问模式 | 说　明 |
|---|---|
| CFile::modeCreate | 调用构造函数构造一个新文件 |
| CFile::modeRead | 打开文件仅供读 |
| CFile::modeWrite | 打开文件仅供写 |
| CFile::modeReadWrite | 打开文件供读写 |
| CFile::typeText | 设置文本文件模式(只能用在子类中) |
| CFile::typeBinary | 设置二进制文件模式(只能用在子类中) |

参数 CFile::typeText 和 CFile::typeBinary 用于设置文本/二进制文件模式，但却只能在派生类中使用，这即是说 CFile 类并未像 C 语言的文件操作函数那样需要设定文件为文本文件还是二进制文件，以决定文件读写时是否可以读出回车换行符。实际上 CFile 类使用的是二进制模式，即可以读出回车换行字符；对于文本文件操作，VC 提供了 CFile 类的子类 CStdioFile，它正是通过设置 CFile::typeText 类来设置文本文件模式的。详细内容请参考其他书籍，在此不再介绍。

参数 pError 为 CFileException 类型，是一个异常类的指针，可通过该类和函数的返回值来确定函数是否调用成功。CFileException 类是异常类的一种，用于检测文件操作中可能出现的错误。关于异常类，请参考其他书籍，在此不再详细介绍。

特别注意：Visual C++ 还替供了一个构造函数用于打开文件，其格式如下：

```
CFile(LPCTSTR lpzsFileName,UINT nOpenFlag)throw(CFileException);
```

可以看出，其定义格式和参数与 Open()函数基本相同。若文件名已知，即可在定义 CFile 类对象的同时打开文件。如：

```
CFile f("D:\\MyVc\\MySdiView.cpp",CFile::modeRead);
```

值得提醒的是构造函数没有返回值，当打开文件出错时只是产生一个异常。所以不能象函数那样通过分支语句来保证文件已打开，而应使用 TRY/CATCH 宏来判断。

例如：

```
TRY
{
   CFile f(strFileName, CFile:: CFile::modeRead);
}
CATCH(CFileException e)
{……
}END_CATCH
```

**2.** 关闭文件函数 **Close**

Close()函数用于关闭由 Open()函数打开的文件。使用 Open()函数打开文件后，应使用 Close()函数释放文件句柄及缓冲区的资源。

值得注意的是，当 CFile 类对象为局部变量时，不调用 Close()函数也不会出问题。这是由于在对象销毁时，析构函数检查文件句柄是否释放，并释放未释放的文件句柄。但建议 Close()函数与 Open()函数成对使用。

**3.** 文件的读/写

CFile 类提供了 Read()函数用于读入文件，定义如下：

```
virtual UINT Read(void * lpBuf,UINT nCount) : throw (CFileException);
```

Read()函数返回值是传输到缓冲区的字节数。

参数 lpBuf：指向用户提供的缓冲区以接收从文件中读取的数据。

参数 nCount：为可以从文件中读出字节数的最大值。UINT 类型等同于 unsigned int，即无符号整型。

注意对所有 CFile 类，如果到达文件尾，则返回值可能比 nCount 小。

CFile 类提供了 Write()函数用于写入文件，定义如下：

```
virtual void Write(void * lpBuf,UINT nCount) : throw (CFileException);
```

Write()函数的参数与 Read()函数的参数类似。

参数 lpBuf：指向用户提供的缓冲区包含将写入文件中的数据。

参数 nCount：从缓冲区内传输的字节数。

Write()在几种情况下均产生异常，包括磁盘满、磁盘为写保护状态等情况。

特别注意：

CFile 类并没有提供类似 EOF 之类的文件结束标志，所以文件的结束是根据 Read()函数的返回值来判断的，Read()函数返回的是实际读出的字符数，当返回值为 0 时，则表示文件已读完。

例如，通过文件的读出和写入操作来实现拷贝的功能。

```
typedef struct {
          char name[10];
          char sex[3];
          int age;
          char tel[14];
```

```
        char school[50];
}STUDENT, *LPSTUDENT;

BOOL MyCopy(CString strSource, CString strTarget)
{
   CFile fs, ft;
   STUDENT  s;
   int  nCount;
   if(! fs.Open(strSource, CFile::modeRead))
   {
        MessageBox("Open Source File Fail !");
        return FALSE
   }
   if(! ft.Open(strTarget, CFile::modeRead))
   {
        MessageBox("Open STarget File Fail !");
        fs.Close();
        return FALSE
   }
   nCount = fs.Read(&s, sizeof(STUDENT));
   while( nCount )
   {
        ft.Write(&s, sizeof(STUDENT));
        nCount = fs.Read(&s, sizeof(STUDENT));
   }
   fs.Close();
   ft.Close();

}
```

以上通过 STUDENT 结构来读写文件，使用 sizeof()函数来确定 STUDENT 结构的大小，当然也可以通过结构的定义来自己计算大小。

但应注意的是，使用结构的时候应保证结构的完整性，否则当 nCount < sizeof(STUDENT)的时候，读入结构 s 中的数据将不完整。若无法保证完整性，应该在写入之前进行判断。

**4.** 文件的定位

CFile 类提供了 Seek()函数用于文件指针的定位。定义如下：

```
virtual LONG Seek(LONG loff, UINT nFrom); throw(CFileException);
```

返回值：如果要求的位置合法，则 Seek()返回从文件开始的字节偏移量。否则值未定义，并产生 CFileException 异常。

参数 loff:　指针移动的字节数。

参数 nFrom：指针移动的参照点，有如下三种定位方式：

（1）CFile::begin　从文件头开始，把指针向后移动 loff 字节。

（2）CFile::curren　从当前位置开始，把指针向后移动 loff 字节。

（3）CFile::end　从文件尾开始把指针向后移动 loff 字节，所以，loff 应为负，如果为正值则超出文件尾。

Seek()函数使用户可以随机访问一个文件的内容，这是通过文件指针移动完成的，指针的移动可分为绝对或相对两种。当文件打开时，文件指针在偏移量为 0 处，即文件开始处。

除了 Seek()函数之外，CFile 还提供了两个实现特殊移动的函数：

```
void  SeekToBegin(); throw  (CFileException);
void  SeekToEnd(); throw  (CFileException);
```

SeekToBegin()函数将文件指针指向文件开始处，等价于 CFile::Seek(0L, CFile::begin)；

SeekToEnd()函数将文件指针指向文件逻辑尾部，等价于 CFile::Seek(0L, CFile::end)。返回值为文件长度(字节数)。

SeekToBegin()、SeekToEnd()和 Seek()函数一样，在定义中都有抛出异常 CFileException 类的语句 throw (CFileException)。这是为了记录函数运行中出现的错误，一般情况下可忽略。

例如，假如 f 为已打开文件，且以 STUDENT 结构(定义见上例)为记录，下面的 MyGetData()函数读取第 nRecord 条记录的值。

```
STUDENT MyGetData(CFile f, int nRecord)
{
   int  Offset;
   STUDENT  s;
   Offset = (nRecord - 1) * sizeof(STUDENT);
   f.SeekToBegin(); //文件指针移到文件头
   f.Seek(Offset, CFile::current);//从当前位置移动 Offset 个字节
   f.Read(&s, sizeof(STUDENT));
   return  s;
}
```

要知道文件指针当前的位置，可使用 GetPosition()函数。该函数返回一个 32 位的无符号整数值，用于标识当前文件指针相对于文件头的偏移量。

例如，下面的程序段可求文件的长度。

```
DWORD GetFileLength(CFile f)
{
   f.SeekToEnd();
   return f.GetPosition();
}
```

**5.** 文件的状态函数

在上节提到了通过 SeekToEnd()函数和 GetPosition()函数来求文件的长度，实际上，文件类提供了专门求文件长度的函数 GetLength()：

```
virtual DWORD GetLength() const; throw (CFileException);
```

该函数没有参数，直接返回文件以字节计的长度。

除此之外 CFile 类还提供了 3 个取得文件名信息的函数，如下所示：

(1) GetFileName()　取得文件名。

(2) GetFileTitle()　取得文件标签。

（3）GetFilePath()　取得文件路径。

以上个函数都没有参数，且返回值都为 CString 型。虽然都用于取得文件名信息，但又各有不同。例如，若当前打开文件为 d:\MYVC\MySdi.data，则 GetFileName()函数取得的值为 MySdi.data；GetFileTitle()函数取得的值为 MySdi；GetFilePath()函数取得的值为 d:\MYVC\MySdi.data。

**6. CFile 类的静态函数**

所谓静态函数是指使用关键字 static 来定义的函数。在这类函数中，不能使用所属类中的任何变量及函数资源。在使用时，也不需要定义所需类的实例，只需按“类名::函数名”的格式直接调用即可。由于该类函数的特殊性，一般用于定义特殊的功能函数。

（1）GetStatus()与 SetStatus()函数

静态函数 GetStatus()用于取得指定文件的状态信息。函数原型为：

```
static BOOL PASCAL GetStatus(LPCSTR lpszFileName,CFileStatus &rStatus);
```

参数 lpszFileName：是 Windows 字符集表示的文件路径，此路径可为绝对路径或相对路径。

参数 rStatus：用于返回文件的信息。这里使用了 CFileStatus 结构来接收状态信息。其中结构定义如下：

```
struct CFileStatus
{
   CTime m_ctime; //文件创建时间
   CTime m_mtime; //文件最后一次修改时间
   CTime m_atime; // 最后一次访问文件并读取的时间
   LONG m_size; //文件逻辑长度
   BYTE m_attribute; //文件属性字节
   TCHAR m_szFullName[MAX_PATH];//Windows 字符集表示的全文件名
};
```

该函数用于测试已存在文件的访问权限十分有用。

与之相对的是静态函数 SetStatus()，用于设置指定文件的状态信息。函数原型为：

```
static BOOL PASCAL SetStatus(LPCSTR lpszFileName,const CFileStatus &rStatus);
```

与 GetStatus()函数不同，此函数没有非静态版本。

例如，设置文件为只读：

```
void SetReadOnly(CString strFilename)
{
   CDileStatus fs;
   CFile::GetStatus(strFilename,fs); //得到文件的属性
   fs.m_attribute |= CFile::readOnly;//改变文件的属性参数
   CFile::SetStatus(strFilename,fs); //设置文件的属性
}
```

（2）Remove()函数

Remove()函数用于删除指定的文件。函数原型为：

```
static void PASCAL Remove(LPCSTR lpszFileName); throw(CFileException);
```

参数 lpszFileName：表示所要删除文件的路径。

只能删除指定的文件，但不可移去一个目录。如果相关联的文件打开或文件不可移去，则函数产生一个异常。这个函数等价于 DOS 中的 del 命令。

例如，删除文件：

```
CString strFileName = "D:\\MyVc\\test.data";
CFile::Remove(strFileName); //删除指定文件
```

(3) Rename()函数

Rename()函数用于更改指定文件的文件名。函数原型为：

```
static void PASCAL Rename(LPCSTR lpszOldFileName,
                (LPCSTR lpszNewFileName ); throw(CFileException);
```

这个函数等价于 DOS 中的 REN 命令。

例如，更改文件名：

```
CString strSource = "D:\\MyVc\\test.data";
CString strTartget = "D:\\MyVc\\Mytest.data";
CFile::Rename(strSource, strTartget );
```

下面将修改 MySdi 应用程序，添加对数据的存储和装入功能。

### 5.4.2 工具栏的可视化设计

**1.** 在工具栏中添加“打开”、“另存为”按钮

在 Workspace 窗口的资源(ResourceView)列表中，双击“Toolbar”下的“IDR _ MAINFRAME”项，打开工具栏编辑器，然后按图 5.12 所示，利用图形编辑工具，依次在原工具栏上添加编辑“打开”、“另存为”工具按钮。

**2.** 设置工具按钮

双击“另存为”按钮(或选中按钮后按 Enter 键)，VC++6.0 显示“Toolbar Button Properties”对话框，如图 5.13 所示。在“ID”文本编辑框中输入“ID _ FILE _ MYSAVE”，在“Prompt”文本编辑框中输入“将数据存储保存\n 另存为”。其中“\n”之前的内容为程序运行时状态栏上给出的改按钮的提示信息；“\n”之后的内容为鼠标光标移至该按钮时光标下方给出的提示信息。

按此方法，设置工具栏上“打开”按钮的“Toolbar Button Properties”对话框，进行如下设置：

```
ID: ID_FILE_MYOPEN
Prompt: 打开一个现有文档\n 打开
```

其余按钮的设置与此类似，留给读者完成。

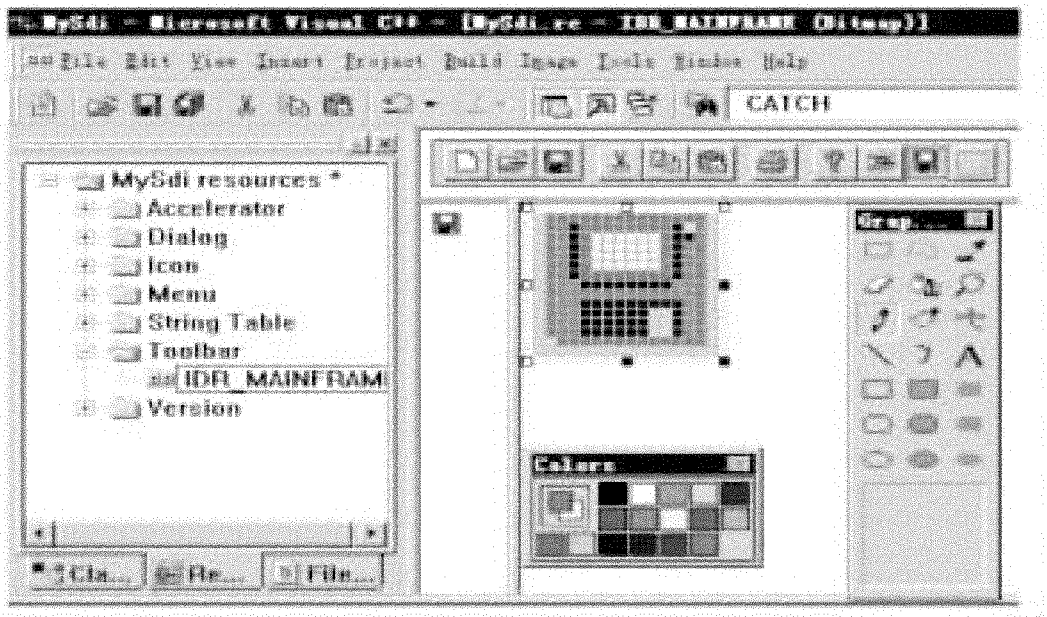



图 5.12　工具栏编辑器


图 5.13　“Toolbar Button Properties”对话框

### 5.4.3　为“打开”按钮编写代码

**1.** 在 **CMySdiView** 类中添加消息响应函数

为工具栏上“打开”按钮的 COMMAND 消息添加响应函数的操作步骤如下：

（1）按 Ctrl + W 键（或选择“View”菜单的“ClassWizard”菜单项），弹出“MFC ClassWizard”对话框。

（2）选择对话框中的“Message Maps”标签，并进行如下选择：

```
Class name:   CMySdiView
Object IDs:   ID_FILE_MYOPEN
Message:      COMMAND
```

（3）单击“Add Function...”按钮来增加新函数，在弹出的“Add Member Function”对话框中，保留默认函数名 OnFileMyopen，单击“OK”按钮，在 CMySdiView 类中为工具栏上“打开”按钮添加消息

响应函数，如图 5.14 所示。

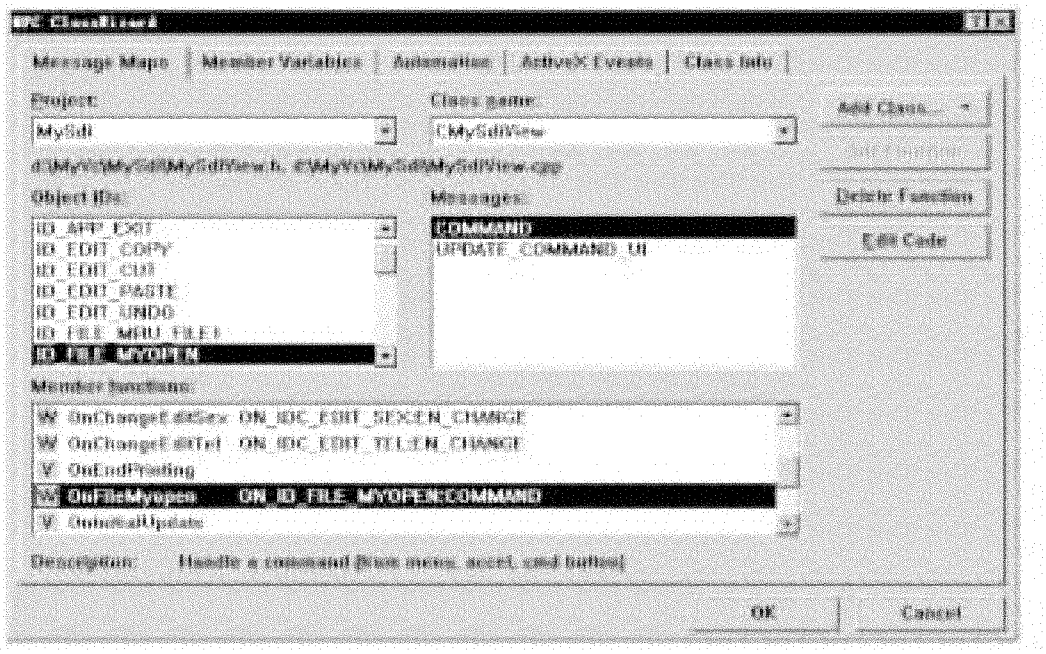



图 5.14 为“ID _ FILE _ MYOPEN”消息添加响应函数 OnFileMyopen

**2.** 编辑消息响应函数代码

在图 5.14 中，单击“Edit Code”按钮（或打开 MySdiView.cpp 文件，定位于 OnFileMyopen 函数），编辑消息响应函数 OnFileMyopen()。

```
void CMySdiView::OnFileMyopen()
{
    // TODO: Add your command handler code here
    CString strFilter = "Dat Files (*.dat)|*.dat|All Files (*.*)|*.*||";
    //打开文件对话框
    CFileDialog FileDlg(true, NULL, NULL,
                        OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
                        (LPCSTR)strFilter, this);
    if(FileDlg.DoModal() != IDOK)return;
    CString strFileName = FileDlg.GetPathName();
    CFile f;

    If(! f.Open(strFileName,CFile::modeRead))
    {
        AfxMessageBox("打开文件失败");
        return;
    }
    CMySdiDoc *pDoc = GetDocument();
    //读出文件中的数据，存放到文档类的数据成员中
    f.Read(&m_nCurrentNo,sizeof(int));
    for(int i=0;i<40;i++){
        f.Read(pDoc->m_student[i].name,10);
```

```
            f.Read(pDoc->m_student[i].sex,3);
            f.Read(&(pDoc->m_student[i].age),sizeof(int));
            f.Read(pDoc->m_student[i].tel,14);
            f.Read(pDoc->m_student[i].school,50);
            }
        f.Close();
        //将文档类的数据传递给视类数据成员,并显示
        m_noList.SetCurSel(m_nCurrentNo);
        m_name = pDoc->m_student[m_nCurrentNo].name;
        m_age = pDoc->m_student[m_nCurrentNo].age;
        m_school = pDoc->m_student[m_nCurrentNo].school;
        m_sex = pDoc->m_student[m_nCurrentNo].sex;
        m_tel = pDoc->m_student[m_nCurrentNo].tel;
        UpdateData(false);
    }
```

### 5.4.4　为“另存为”按钮编写代码

类似于“打开”按钮，为工具栏上“另存为”按钮的 COMMAND 消息添加消息响应函数 OnFileMysave()。并在其中添加代码：

```
void CMySdiView::OnFileMysave()
{
    CString strFilter = "Dat Files(*.dat)|*.dat|All Files(*.*)|*.*||";
    CFileDialog FileDlg(false, NULL, NULL,
                        OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
                        (LPCSTR)strFilter, this);
    if(FileDlg.DoModal() != IDOK)return;
    CString strFileName = FileDlg.GetPathName();
    CFile  f;
    if(! f.Open(strFileName,CFile::modeCreate | CFile::modeWrite))
    {
        AfxMessageBox("创建文件失败");
        return;
    }
    CMySdiDoc *pDoc = GetDocument();
    f.Write(&m_nCurrentNo,sizeof(int));
    for(int i=0;i<40;i++){
        f.Write(pDoc->m_student[i].name,10);
        f.Write(pDoc->m_student[i].sex,3);
        f.Write(&(pDoc->m_student[i].age),sizeof(int));
        f.Write(pDoc->m_student[i].tel,14);
        f.Write(pDoc->m_student[i].school,50);
}
    f.Close();
}
```

### 5.4.5 查看结果

编译、连接运行 MySdi 程序，操作步骤如下：

（1）选择“Build”菜单中的“Build MySdi.exe”菜单项，Visual C++ 6.0 就会编译并连接 MySdi.exe 程序。

（2）选择“Build”菜单中的“Execute MySdi.exe”菜单项，Visual C++6.0 就会执行 MySdi.exe 程序，如图 5.1 所示的 MySdi.exe 程序主窗口也随之出现。

（3）编辑录入数据，单击工具栏上“另存为”按钮，将弹出另存为文件对话框，选择输入适当的文件名，就可把学生档案数据保存在文件中。

（4）单击工具栏上“打开”按钮，将弹出打开文件对话框，选择刚存储的文件，程序就将学生档案数据显示在对应的编辑框中。可以看到程序具有了数据录入和浏览功能。

（5）单击 MySdi.exe 程序主窗口右上角关闭按钮，退出 MySdi.exe 程序。

## 5.5 添加串行化功能

除了利用 CFile 类进行文件读写操作外，本节将使用串行化对文档类数据进行存储和装入，为 MySdi 程序添加串行化功能。

### 5.5.1 串行化概述

一个对象通过某个操作，在程序退出时可被存储起来，当程序启动时又可被恢复，对象的这种存储和恢复处理过程在 MFC 中称为“串行化（Serialization）”。在 Visual C++ 中，负现这重要任务的是 CObject 类所具有的 Serialize()成员函数。所有 CObject 派生类都可以利用 CObject 的串行化操作。

串行化的基本思想是一个类对象能够将它记录在成员变量中的当前状态存储起来，之后也可以通过串行化的读取恢复操作重新创建对象，对于一个想实现串行化操作的 CObject 派生类对象而言，必须完善 Serialize()成员函数的功能。

MFC 使用 CArchive 类对象作为执行串行化对象和存储介质之间的中继，这个对象总是和一个包含有必需的文件信息的 CFile 对象相关联，这个 CFile 对象中实际打开了记录信息的某个文件，执行串行化操作的对象可以使用 CArchive 对象去读或写，不必考虑存储介质上的文件情况。

下面分别对 CArchive 类和函数 Serinlize()进行介绍。

**1. CArchive 类**

CArchive 类没有基类，它提供了串行化对象从文件中读写的类型安全缓冲机制，可以把 CArchive 对象想象成一种二进制流，就像输入/输出流一样可以顺序高效的处理二进制对象数据。

使用 CArchive 对象之前，必须先创建一个 CFile 对象，同时保证 CArchive 对象的读写标志设置和文件打开方式相一致。对于一个 CArchive 对象，可以进行存储操作，也可以读取，但不能两者同时进行。

（1）CArchive 类操作符“ << ”和“ >> ”

操作符“ << ”和“ >> ”直接支持表 5.8 中列出的简单数据类型和 CObject 派生类。

表 **5.8**　**CArchive** 类的 >> 和 << 操作符支持类型

| | | | |
|---|---|---|---|
| CObject * | SIZE 和 CSize | float | WORD |
| CString | POINT 和 CPoint | DWORD | BYTE |
| LONG | RECT 和 CRect | CTime 和 CTimeSpan | double |
| int | COleDateTime | | |

CArchive 对象的引用，类似于流的输入/输出（cin/cout），可以使用一系列的操作符以简化程序，例如：

```
int  x;
CString  y;
……
ar << x << y;
```

（2）CArchive 类的成员函数，如表 5.9 所示。

表 **5.9**　**CArchive** 类的成员函数

| 成员函数 | 功能 |
|---|---|
| Read、Write | 读写指定字节数的缓冲区内容 |
| ReadString、WriteString | 读写一行文本 |
| ReadObject、WriteObject | 调用一个对象的 Serialize 函数来读或写 |
| ReadClass、WriteClass | 读写一个 CRuntimeClass 指明对象 |
| IsLoading、IsStoring | 判断当前读写状态 |

**2. Serialize( )函数**

在“MFC AppWizard”自动生成的程序框架中，文件的串行化操作都是由 CDocument 派生类的成员函数 Serialize( )完成的，它的结构如下：

```
void CMySdiDoc::Serialize(CArchive& ar)
{
    if (ar.IsStoring())
    {
        // TODO: add storing code here
    }
    else
    {
        // TODO: add loading code here
    }
}
```

在这个成员函数中，使用 CArchive 对象完成具体的操作。参数中传递进来的 CArchive 引用对象，是由 MFC 程序框架根据用户输入需要执行串行化操作时创建的，它包含了所需要的文件的信息，使用它可以对各种 CArchive 类支持的数据格式进行读写、调用其他 CObject 派生类对象的串行化函数等。

“MFC AppWizard”自动生成的代码已经完成的工作是：用户选择“打开”、“保存”或“另存为”等命令时，程序框架创建着这个文件的 CFile 对象，将它关联到新创建的 CArchive 对象上，并设置 CArchive 对象的“Store”或“Load”标志，用这个对象来调用 CDocument 派生类的 Serialize( )成员函数。在 Serialize( )函数完成读写操作返回后，自动删除 Serialize( )函数和 CFile 对象。

### 5.5.2 添加串行化存储和装入

打开 MySdiDoc.cpp 文件，并定位到 Serialize( )函数。如前所述，当用户从程序的文件菜单中选择“保存”、“另存为”或“打开”时，函数 Serialize( )会自动执行。Serialize( )函数的参数 ar 代表正在读或写的文件。

如果从文件菜单中选取“保存”、或“另存为”，则将执行 Serialize( ) 函数中 if 条件后的语句；如果选取“打开”，则将执行 else 后的语句。因此，所需做的工作就是：

在 if 后面添加用于把数据写到文件中的代码；在 else 后添加用于从文件读取数据的代码。

添加代码后的 Serialize( )函数为：

```
void CMySdiDoc::Serialize(CArchive& ar)
{
    if(ar.IsStoring())
    {
        // TODO: add storing code here

        for(int i = 0;i < 40;i ++ )
        {
        ar.Write(m_student[i].name,10);
        ar.Write(m_student[i].sex,3);
        ar < < m_student[i].age;
        ar.Write(m_student[i].tel,14);
        ar.Write(m_student[i].school,50);
        }
    }
    else
    {
        // TODO: add loading code here

        for(int i = 0;i < 40;i ++ )
        {
        ar.Read(m_student[i].name,10);
        ar.Read(m_student[i].sex,3);
        ar < < m_student[i].age;
        ar.Read(m_student[i].tel,14);
        ar.Read(m_student[i].school,50);
        }
```

```
        }
    }
```

### 5.5.3　查看结果

（1）选择“Build”菜单中的“Build MySdi.exe”菜单项，Visual C++6.0就会编译并连接MySdi.exe程序。

（2）选择“Build”菜单中的“Execute MySdi.exe”菜单项，Visual C++6.0就会执行MySdi.exe程序，如图5.1所示的MySdi.exe程序主窗口也随之出现。

（3）除了前面功能外，添加了串行化功能。其操作方法是：

选择“文件”菜单下的“打开”或“保存”子菜单项，程序提供常规的文件“打开”或“另存为”功能。

（4）单击MySdi.exe程序主窗口右上角关闭按钮，退出MySdi.exe程序。

## 习　题　五

1. 修改MySdi程序，要求：

（1）删除多余的菜单和工具栏按钮；

（2）添加一个数据存取菜单，其中包含“打开”和“另存为”菜单项，完成工具栏“打开”和“另存为”按钮相同的功能。

2. 完善MySdi程序，添加系统管理员权限，只有系统管理员才能编辑录入数据，一般用户仅限于浏览操作（提示：用“口令”对话框和编辑框的属性进行控制，参考第4章表4.10 CEdit类的基本方法）。

# 第6章

# 设备环境与屏幕绘图

## 本章导读

图形化是计算机人机界面的重要组成部分，在程序设计中，掌握图形设计是非常重要的。例如，在屏幕上画一个简单的图形，就需要了解相应的图形函数。

本章将介绍 MFC 中一些用于绘图、文字处理的专用类。同时，通过编写一个简单的绘图程序，使读者掌握以下内容：

- 了解设备环境和设备环境类
- 使用设备环境类及图形设备接口（GDI）对象进行绘图
- 工具栏的设计
- 菜单设计
- 快捷菜单
- 设计鼠标形状

---

## 6.1 绘图程序

图 6.1 所示是 MyDraw 绘图程序的运行结果，该程序具有如下功能：

（1）绘制“直线”、“矩形”、“圆角矩形”和“椭圆”基本图形，以及图形填充，并用不同的鼠标光标来标识当前选择的绘图类型。

（2）能设置画笔的粗细（线宽）和绘图颜色。

（3）提供菜单、工具栏按钮和快捷键等选择方式。

（4）能通过快捷菜单选择要绘制的基本图形。

本章将按知识点进行目标分解，按以下步骤完成该程序的开发：

（1）编写提供工具栏按钮选择方式的绘图程序。

（2）增加菜单和快捷键选择方式，改善人机交互。

（3）添加快捷菜单完善程序。

下面，首先介绍有关绘图的基本知识：设备环境和设备环境类，图形设备接口（GDI）对象和基

本矢量图形。

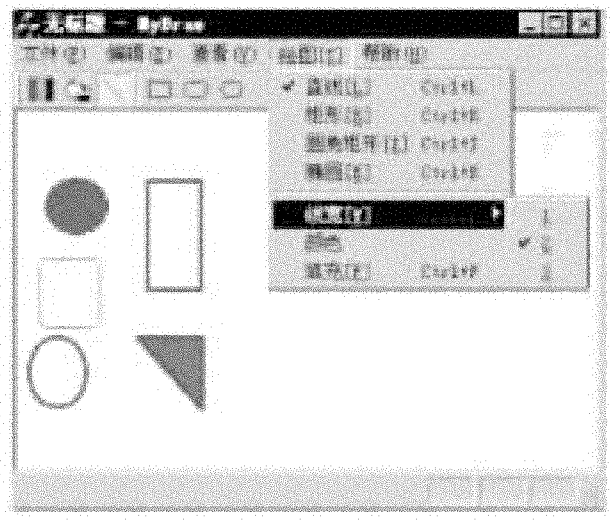


图 6.1　MyDraw 绘图程序的运行情况

## 6.2　设备环境和设备环境类

### 6.2.1　设备环境的概念

设备环境也称设备上下文(Device Context，简称 DC)，是计算机物理设备的代表，也是图形设备接口的主要组成部分。

由于 Windows 是一个与设备无关的操作系统，即 Windows 不允许直接访问硬件，如果用户想将文本和图形绘制到显示器或其他某个设备中去，必须通过“设备环境”这个抽象层与硬件进行通信，设备上下文对象的作用就是实现 Windows 的设备无关性，任何向屏幕上进行输出的功能都要间接地通过它来完成。

设备上下文是 Windows 的一种数据结构，它包含了有关如显示器或打印机等设备的绘画属性信息。所有的绘画都是通过设备上下文对象来实现的，该对象封装了 Windows 的画线、图形和文本的 API 函数。设备上下文允许在 Windows 下的独立于设备的绘画。设备上下文不仅能够被用来在屏幕上绘画，它也可以将绘画输出到打印机和图元文件中。

### 6.2.2　设备环境类

设备环境类 CDC 直接继承于 CObject 类，该类定义了一类设备对象。CDC 对象提供了非常多的成员函数，与设备环境的显示器、打印机等一起工作。

例如，如果要在显示器等设备上绘制图形，我们可以用 MFC 提供的设备环境类 CDC 类，因为 CDC 类中包含了绘图所需要的所有成员函数。

同时 MFC 还提供了几个 CDC 的派生类：CPaintDC 和 CClientDC，以供特殊用途。其中

CClientDC 的对象用于管理窗口的用户区，即窗口中不含工具条、状态栏和滚动条的区域。CPaintDC 类封装了 Windows 的通用习惯用法：调用 BeginPaint 函数，然后在上下文中绘画，再调用 EndPain 函数。设备环境类 CDC 及其派生类如图 6－2 所示。

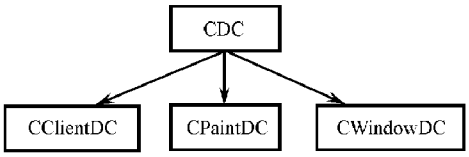



图 6.2 CDC 与其子类继承图

**1. CDC 基类**

CDC 类是其他 DC 类的基类，CDC 类封装了使用设备环境的各种 GDI 函数，它用于定义一个设备环境对象，并提供了在显示器、打印机和窗口的客户程序区域上画图的方法。

（1）在视图类的 OnDraw( )函数中绘图

如图 6－3 所示，在视图类的 OnDraw 函数中绘图时，直接使用 OnDraw( )函数中的 CDC 形参指针 pDC，调用它的函数进行绘图。

```
void CMyHelloView::OnDraw(CDC * pDC)
{
CMyHelloDoc * pDoc = GetDocument();
ASSERT_VALID(pDoc);
// TODO: add draw code for native data here

pDC->SelectStockObject(LTGRAY_BRUSH);                    //选择刷子

CRect rect(10,10,200,70);                                //定义一个矩形
pDC->Rectangle( rect);                                   //画一个矩形
pDC->TextOut(100,80,"Hello，我们开始 VC++ 编程了!");     //绘制文本
}
```

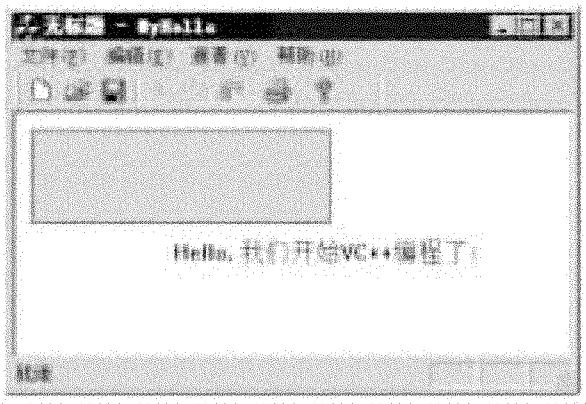


图 6.3 CDC 简单应用：画一个矩形、绘制文本

（2）在视图类的一般函数中绘图

在视图类的一般函数中绘图时，可采用如下两种方法进行绘图。

第一种是在堆栈上构造对象，构造的对象会被自动删除。程序实现如下：

```
void CMyHelloView::OnLbuttonDown(UINT nFlags, CPoint point)
{
    CDC    dc(this);
    CRect    rect(0,0,100,100);      //定义一个矩形
    dc.Rectang(rect);
}//dc 自动地被释放了
```

第二种是调用 GetDC()函数获得显示设备环境 DC 的指针 pDC 进行绘图，但绘图完毕后，必须调用 ReleaseDC 函数来释放它。程序实现如下：

```
void CMyHelloView::OnLbuttonDown(UINT nFlags, CPoint point)
{
    CRect rect;
    CDC * pDC = GetDC( );
    pDC -> Rectang(rect);
    ReleaseDC(pDC);  // 释放 pDC
}
```

**2. CPaintDC 类**

CPaintDC 对象代表了一个窗口的绘图画面，主要用来绘图。它可以用来处理来自 Windows 的 WM _ PAINT 消息。改变窗口大小或移动覆盖在窗口上的窗口或对话框时，Windows 会发送 WM _ PAINT 消息以通知客户区的变动，而 WM _ PAINT 消息的处理是在 OnPaint()消息处理函数中进行的。使用 CPaintDC 类绘图的步骤是：

（1）创建一个 CPaintDC 的对象：CPaintDC dc(this)。

（2）调用 CPaintDC 的函数进行绘图。

（3）撤消该 CPaintDC 的对象。

例如，OnPaint()函数中添加画线、画椭圆的代码为：

```
void MyTestDlg::OnPaint()
{//生成一个 CPaintDC 类的实例 dc，用于调用它的函数进行绘图
   CPaintDC dc(this);
   dc.MoveTo(10,10);                //调用 MoveTo 函数，定点到点(10,10)
   dc.LineTo(100,100);              //调用 LineTo 函数，画线到点(100,100)
   dc.Ellipse(120,120,160,160);     //调用画椭圆函数 Ellipse
   CDialog::OnPaint();
}
```

**3. CClientDC 类**

CClientDC 对象用来自动处理对描述窗口的客户程序区域的设备环境进行调用和释放。在 CClientDC 对象创建时自动调用 GetDC()，在撤消时将自动调用 ReleaseDC()。

例如，在客户区画一条直线的代码如下：

```
void CMainWnd::OnLbuttonDown(UINT nFlags, CPoint point)
{
```

```
    CClientDC dc(this);
    CRect rect;
    GetClientRect(&rect);
    dc.MoveTo(rect.left,rect.Top);          //调用 MoveTo 函数,定点到客户区的左上角
    dc.LineTo(rect.right,rect.bottom);      // 调用 LineTo 函数,画线到客户区的右下角
}
```

## 6.3 图形设备接口(GDI)对象

GDI(Graphic Device Interface)对象与设备环境对象的关系,类似于笔和纸的关系。也就是说,GDI 提供了用于在 DC 上画图的绘图工具。MFC 的 GDI 中包含了各种绘图类并提供各类的绘图函数,即定义了若干种对于 Windows 的绘工具的图形对象,该类的继承关系如图 6.4 所示,它们包括:

- CPen(画笔类)
- CBrush(画刷类)
- CFont(字体类)
- CBitmap(位图类)
- CPalette(调色板类)
- CRgn(绘图区域类)

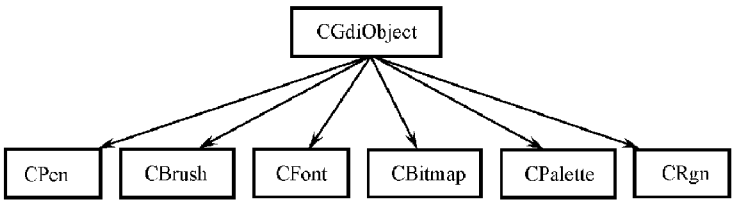



图 6.4 MFC 的 GDI 类继承图

使用 MFC 程序中的图形设备接口(GDI)对象,通常按以下步骤操作:

(1) 定义一图形对象,并用对应的 Create * 方法对该对象进行初始化。例如,如要创建一个 CPen 对象,请使用 CreatePen()方法。

(2) 一般使用 CDC::SelectObject()方法选定放入当前设备环境的新对象。该方法返回一个指向被替换的对象的指针。

(3) 在选定图形对象后,用各种画图函数画图。完成后,再次使用 CDC::SelectObject()方法,选定被替换的图形对象,重新放入设备环境,使一切恢复原样。

### 6.3.1 画笔:CPen 类

CPen 类封装了一个 Windows GDI 画笔,并且提供了用于操作 CPen 对象的若干方法。CPen 类用来决定画线的风格和颜色。

在使用画笔之前,首先必须构造画笔对象,有两种方法:

（1）构造和初始化对象都在带参数的构造函数中一步完成，如：

```
CPen  newPen(PS_SOLID,2, RGB(0,255,0));//生存一支颜色为绿色、
                                        //宽度为2个像素的实心画笔
```

（2）构造和初始化分两步完成，构造由不带参数的构造函数完成，而初始化由初始化函数完成，如：

```
CPen newPen;
newPen.CreatePen(PS_SOLID,2,RGB(0,255,0));
```

例如，用一支颜色为绿色、宽度为2个像素的实心画笔画线和画椭圆的程序段如下：

```
CDC* pDC = GetDC();     //获得显示设备环境DC的指针pDC
CPen newPen;
newPen.CreatePen(PS_SOLID,2,RGB(0,255,0));
CPen* poldpen = pDC->SelectObject(&newPen);//将创建的画笔选入内存DC,同时
                                           //暂时挤出并保存原画笔
pDC->MoveTo(10, 10);                  //调用MoveTo函数,定位于点(10,10)
pDC->LineTo(100,100);                 // 调用LineTo函数,画线至点(100,100)
pDC->Ellipse(120,120,160,160);        // 调用画椭圆函数Ellipse

pDC->SelectObject(poldpen);           //恢复内存DC中原有的画笔
ReleaseDC(pDC);                       //释放显示DC
```

特别注意：

默认的画笔为黑色、一个像素宽度、实心，默认的画刷为白色，例如，6.2.2中的OnPaint()函数中没有创建画笔，则用默认的画笔绘图。

### 6.3.2　画刷：CBrush类

CBrush类封装了一个Windows GDI画刷，并且提供了用于操作CBrush对象的若干方法。画刷可设置画刷的色彩和区域填充的绘图方法。在使用画笔、画刷之前，首先必须构造画笔、画刷对象。有两种构造方法：

（1）构造和初始化对象都在带参数的构造函数中一步完成，如：

```
CBrush newBrush(RGB(0,0,255));     //蓝色画刷
```

（2）构造和初始化分两步完成，构造由不带参数的构造函数完成，而初始化由初始化函数完成，如：

```
CBrush newBrush ;
newBrush.CreateSolidBrush(RGB(0,0,255));
```

例如，用一支颜色为绿色、宽度为2个像素的实心画笔画椭圆，并用蓝色画刷填充椭圆内部的程序段如下(如果不填充椭圆内部，则相关的画刷语句省略)：

```
CDC* pDC = GetDC();     //获得显示设备环境DC的指针pDC
CPen newPen;
newPen.CreatePen(PS_SOLID,2,RGB(0,255,0));
```

```
CBrush newBrush ;
newBrush.CreateSolidBrush(RGB(0,0,255));
CPen * poldpen = pDC -> SelectObject(&newPen);//将创建的画笔选入内存 DC,
                                              //同时暂时挤出并保存原画笔
CBrush * poldbrush = pDC -> SelectObject(&newBrush); //将创建的画刷选入内存 DC,
                                                     //同时暂时挤出并保存原画刷
pDC -> MoveTo(10, 10);                  //调用 MoveTo 函数,定点到点(10,10)
pDC -> LineTo(100,100);                 // 调用 LineTo 函数,画线到点(100,100)
pDC -> Ellipse(120,120,160,160);        // 调用画椭圆函数 Ellipse
pDC -> SelectObject(poldpen);           //恢复内存 DC 中原有的画笔
pDC -> SelectObject(poldbrush);         //恢复内存 DC 中原有的画刷
ReleaseDC(pDC);                         //释放显示 DC
```

### 6.3.3 字体:CFont 类

CFont 对象封装了一种 Windows GDI 字体,并且提供了用于操作 CFont 对象的若干方法。CFont 类用来决定绘图文本时的字体,要使用字体,必须先创建字体,然后将其选进要进行文本输出的 DC,就可以利用文本输出函数显示该字体形式的文本内容了。

创建字体由两步完成:

(1) 定义一个 CFont 类的对象,应用框架会调用构造函数,如:

```
CFont myFont;
```

(2) 调用 CFont 类的创建字体函数,从而将构造的 CFont 对象与 Windows 的某种字体相关联。

```
myFont.CreateFont(……);
```

其中 CreateFont()的原形为:

```
BOOL CreateFont(int nHeight,          //字体高度
        int nWidth,                   //字符平均宽度
        int nEscapement,              //文本行角度
        int nOrientation,             //字符角度
        int nWeight,                  //字符粗细度
        BYTE bItalic,                 //斜体
        BYTE bUnderline,              //下划线
        BYTE cStrikeOut,              //删除线
        BYTE nCharSet,                //字符集
        BYTE nOutPrecison,            //字体输出结果和要求的匹配程度
        BYTE nClipPrecison,           //如何裁剪落于裁剪区之外的字符
        BYTE nQuality,                //字体属性匹配的精确程度
        BYTE nPitchAnFamily,          //字体间距和字体簇
        BYTE lpszFacename             //字体名称
        );
```

例如,图 6.5 所示是利用了字体显示函数 FontOut()的程序的运行结果。

```
void CVc09View::OnDraw(CDC * pDC)
```

```
{
    CVc09Doc * pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    pDC->SetBkColor(RGB(240,240,250));          //设置背景颜色
    pDC->SetTextColor(RGB(255,0,0));            // 设置文本颜色

    int ny = -5;
    int ndl = 0;
    for (int i = 24; i >= 12; i -= 4) {
        FontOut(pDC, ny, i, ndl);
        ndl += 300;
    }
}

void CVc09View::FontOut(CDC * pDC, int& nHeight, int nPoints,int dline)
{
    TEXTMETRIC textM;                           //定义文本结构 TEXTMETRIC 变量
    CFont font;                                 //定义 CFont 类的对象
    CString str;
    //创建字体
    font.CreateFont(-nPoints,0, dline, 0, 400, FALSE, FALSE, 0,
                    ANSI_CHARSET, OUT_DEFAULT_PRECIS,
                    CLIP_DEFAULT_PRECIS, DEFAULT_QUALITY,
                    DEFAULT_PITCH | FF_SWISS,"宋体");
        //将创建的字体选入内存 DC,同时暂时挤出并保存原字体
    CFont * poldfont = (CFont * ) pDC->SelectObject(&font);
    pDC->GetTextMetrics(&textM);                //获取文本信息
    str.Format("这是%d点阵宋体字", nPoints);
    pDC->TextOut(10, nHeight, str);
    nHeight -= textM.tmHeight + textM.tmExternalLeading;
    pDC->SelectObject(poldfont);                //恢复内存 DC 中原有的字体
}
```

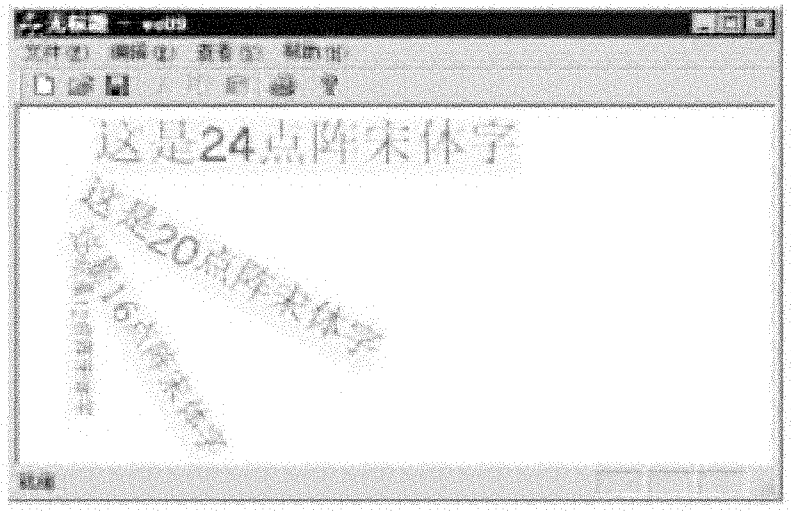



图 6.5　字体显示程序 Vc09.exe 的运行界面

# 6.4 矢 量 图 形

Windows GDI 为应用程序提供了许多类型的矢量图形的输出函数。用这些函数可以画出丰富多彩的图形。

本节将介绍如何利用 DGI 对象和输出函数进行画图。

## 6.4.1 绘图模式

绘图模式决定了当前画上去的颜色与画面上已存在的颜色之间的组合关系。设置绘图模式的函数原型如下：

```
CDC::SetRop2(int nDrawMode);
```

其中参数 nDrawMode 为绘图模式标识符，可取表 6.1 中所列的值。

表 **6.1** **Windows** 定义的绘图模式

| 图模式标识符 | 描　　述 |
|---|---|
| R2 _ BLACK | 像素颜色总是黑色 |
| R2 _ COPYEN | 像素是画笔的颜色 |
| R2 _ MASKNOTPEN | 像素颜色是屏幕颜色与画笔颜色的反色(最后像素 =[NOT 画笔]AND 屏幕像素) |
| R2 _ MASKREN | 像素颜色是屏幕颜色与画笔颜色(最后像素 = 画笔 AND 屏幕像素) |
| R2 _ MASKRENNOT | 像素颜色是屏幕颜色的反色与画笔颜色(最后像素 = 画笔 AND[ NOT 屏幕像素]) |
| R2 _ MERGENOTPEN | 像素颜色是屏幕或画笔颜色的反色(最后像素 =[NOT 画笔]OR 屏幕像素) |
| R2 _ MERGEPEN | 像素颜色是屏幕或画笔颜色(最后像素 = 画笔 OR 屏幕像素) |
| R2 _ NOP | 像素颜色保持不变 |
| R2 _ NOT | 像素颜色是屏幕颜色的反色 |
| R2 _ NOTCOPYPEN | 像素颜色是画笔颜色的反色 |
| R2 _ WHITE | 像素颜色始终为白色 |
| R2 _ XORPEN | 像素颜色是画笔颜色异或屏幕颜色(最后像素 = 画笔 XOR 屏幕像素) |

## 6.4.2 基本矢量图形

**1.** 点

MFC 用 CPoint 类将 POINT 结构封装起来，CPoint 类提供了多种运算符，使点的计算变得更加容易。

画点的函数为 CDC::SetPixel()，其原型为：

```
COLORREF SetPixel(POINT point, COLORREF crColor);
```

其中 point 用于设定要画的点的坐标，crColor 用于设定颜色。

例如，在屏幕(100,200)处画一红色点。程序实现如下：

```
……
CClientDC dc(this);
CPoint point(100,200);
dc.SetPixel( point,RGB(255,0,0));
……
```

**2.** 直线

线条的绘制是由 CDC::MoveTo()和 CDC::LineTo()函数完成。

MoveTo()函数用于设定当前点的位置，其原型为：

```
CPoint MoveTo(int x, int y);
CPoint MoveTo(POINT point);
```

LineTo()函数用于画一条从当前点到指定点的直线，然后将指定点设为当前点。画线的样式取决于选定的画笔。LineTo 函数的原型为：

```
BOOL LineTo(int x, int y);
BOOL LineTo(POINT point);
```

例如，在屏幕上从点 p1(100,200)到点 p2(150,300)画线段。程序实现如下：

```
……
CClientDC dc(this);
CPoint p1(100,200),p2(150,300);
dc.MoveTo(p1);
dc.LineTo(p2);
……
```

**3.** 矩形

MFC 将 RECT 结构封装在 CRect 类中，凡能用 RECT 结构的地方都可以用 CRect 代替。由于 CRect 提供了一些成员函数和重载运算符，使得 CRect 的操作更加方便，详见 6.8.2 节 CRect 类的有关内容。

画矩形常用的函数是 CDC::Rectangle()和 CDC::FillRect()。

其中 CDC::Rectangle()的原型为：

```
BOOL Rectangle(int x1, int y1, int x2, int y2);
BOOL Rectangle(LPCRECT lpRect);
```

CDC::FillRect()的原型为：

```
void FillRect(LPCRECT lpRect, CBrush * pBrush);
```

该函数用给定的画刷给矩形填充颜色。

例如，在屏幕上画一个左上角为(120,120)，右下角为(160,160)的矩形。程序实现如下：

```
……
CPaintDC dc(this);
dc.Rectangle(120,120,160,160);
……
```

**4**. 椭圆

画椭圆的函数为 CDC::Ellipse( ),它是以指定的矩形区域画一个椭圆。其函数原型为:

```
BOOL Ellipse(int x1, int y1, int x2, int y2);
    BOOL Ellipse(LPCRECT lpRect);
```

它的参数用于确定一个矩形区域。

例如,在屏幕上左上角为(120,120),右下角为(160,160)的矩形内画一个椭圆。程序实现如下:

```
……
CPaintDC dc(this);
dc.Ellipse(120,120,160,160);
……
```

## 6.5 绘图程序

有了前面的预备知识,下面开始编写提供工具栏按钮选择方式的绘图程序 MyDraw,它的运行情况如图 6.6 所示。

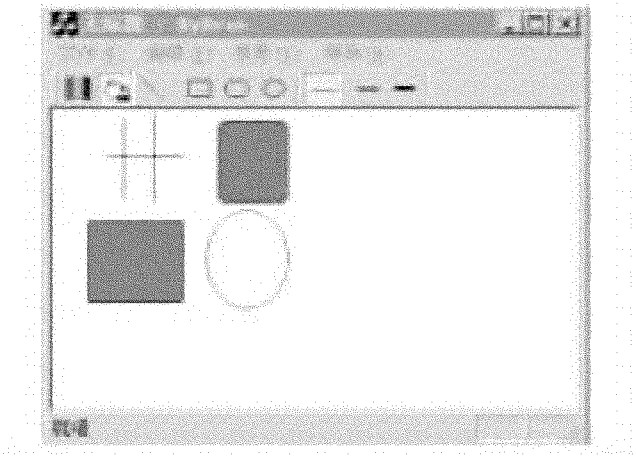


图 6.6 绘图程序的运行情况

### 6.5.1 创建绘图程序工程

下面是创建 MyDraw 工程的详细步骤:

（1）启动 Visual C++6.0。从“File”菜单中选择“New”。此时，Visual C++将显示一个“New”对话框。

（2）在“New”对话框中选择“Project”标签 。然后指定工程类型 MFC AppWizard[exe]、工程名 MyDraw 和工程位置 d:\MYVC。

（3）点击“OK”按钮，弹出“MFC AppWizard - Step 1”对话框，选择“Single document”单选按钮，创建一个基于单文档(SDI)的应用程序，如图 6.7 所示。

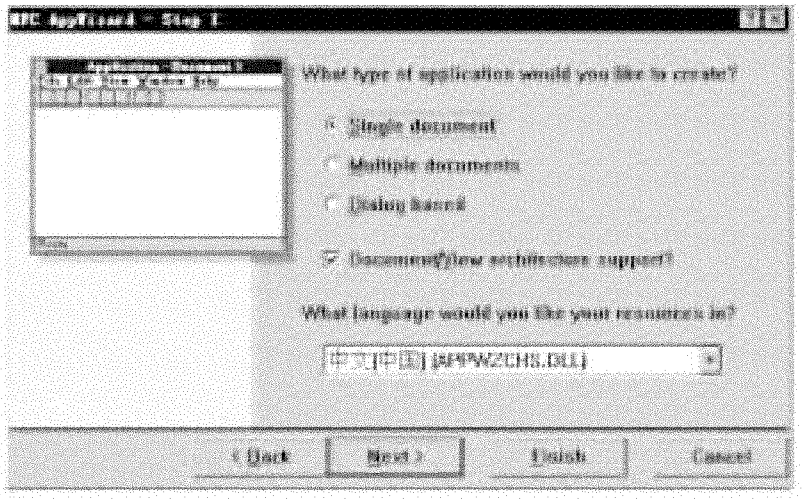



图 6.7　“MFC AppWizard - Step 1”对话框

（4）在“MFC AppWizard - Step 1”对话框中，单击“Finish”按钮。此时 Visual C++将显示“NewProject Information”窗口。

（5）单击“OK”。于是，Visual C++就会创建 MyDraw 工程以及相关的所有文件。

### 6.5.2　工具条的可视化设计

**1.** 编辑绘图工具按钮

在“Workspace”窗口的资源(ResourceView)列表中，双击“Toolbar”下的“IDR _ MAINFRAME”项，同时打开“Graphics”和“Colors”绘画工具箱。

删除其中本例不需要的工具按钮，方法是拖动需删除的按钮到工具栏编辑区之外。然后依次新建“颜色”、“填充区域”、“画线”、“画矩形”、“画圆角矩形”、“绘制椭圆”、“线宽一”、“ 线宽二”和“线宽三”等工具按钮，设计好的工具栏如图 6.8 所示。

**2.** 设置工具按钮

双击“颜色”按钮(或选择颜色按钮后按 Enter 键)，VC++6.0 显示“Toolbar Button Properties”对话框，如图 6.9 所示。在“ID”文本编辑框中输入“ID _ COLOR”，在“Prompt”文本框中输入“选择将要使用的颜色\n 选颜色”。

特别说明：

在“Prompt”文本框中，其中，“\n”之前的内容为程序运行时状态栏上给出的该按钮的提示信息；“\n”之后的内容为鼠标光标移至该按钮时光标下方给出的提示信息。

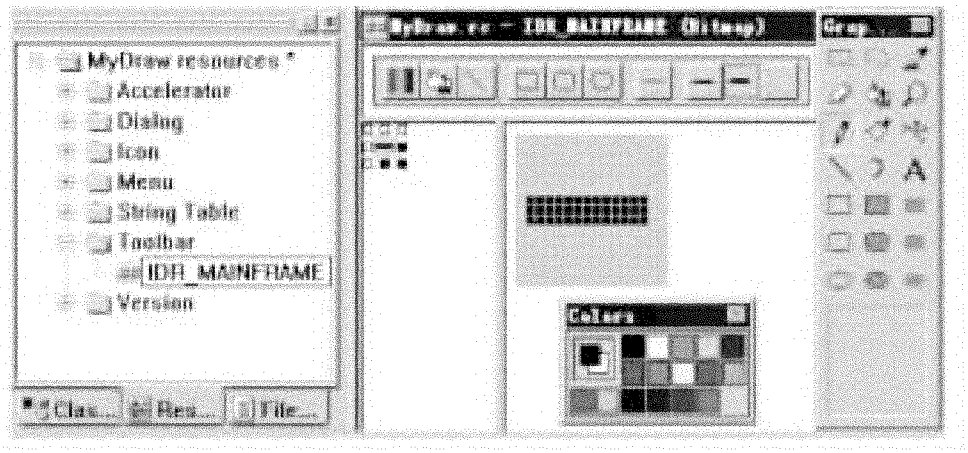



图 6.8 编辑工具栏按钮

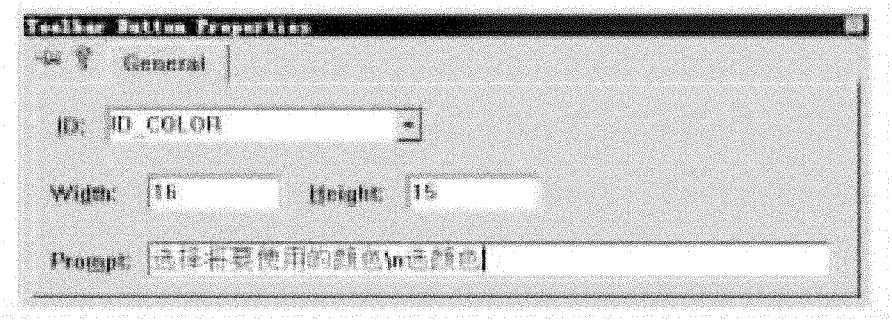



图 6.9 “Toolbar Button Properties”对话框

用同样的方法,设置好表 6.2 所列的工具栏上其他按钮的属性对话框。

表 **6.2** 工具栏按钮的属性表

| 工具栏按钮 | ID 号 | Prompt |
|---|---|---|
| 选颜色 | ID _ COLOR | 选择将要使用的颜色 \ n 选颜色 |
| 填充区域 | ID _ FILL | 区域填充 \ n 填充 |
| 画线 | ID _ LINE | 画直线 \ n 直线 |
| 画矩形 | ID _ RECT | 画矩形 \ n 矩形 |
| 画圆角矩形 | ID _ ELLIRECT | 画圆角矩形 \ n 圆角矩形 |
| 绘制椭圆 | ID _ ELLIPSE | 画椭圆 \ n 椭圆 |
| 线宽一 | ID _ WIDTH1 | 线宽为一个像素 \ n 线宽为一 |
| 线宽二 | ID _ WIDTH2 | 线宽为二个像素 \ n 线宽为二 |
| 线宽三 | ID _ WIDTH3 | 线宽为三个像素 \ n 线宽为三 |

### 6.5.3 声明 CMyDrawView 类的数据成员

(1) 添加数据成员

在“Workspace”窗口中选择“ClassView”，右击其中的“CMyDrawView”类。VC++6.0显示快捷菜单，如图6.10所示。

选择“Add Member Variable”选项，VC++6.0显示“Add Member Variable”对话框，如图6.11所示。进行如下设置：

```
Variable Type: int
Variable Name: m_type
Access: Public
```

单击“OK”按钮，就在CMyDrawView类的公有段(Public)定义了一个int型变量m_type。

按以上方法，在CMyDrawView类中添加表6.3所列成员变量。

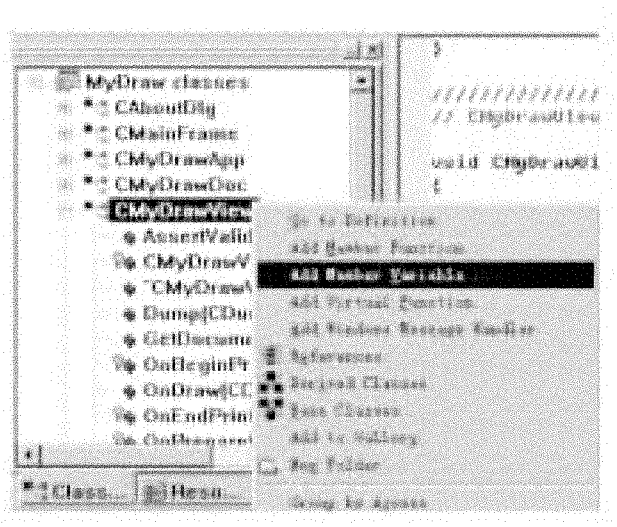


图6.10　快捷菜单

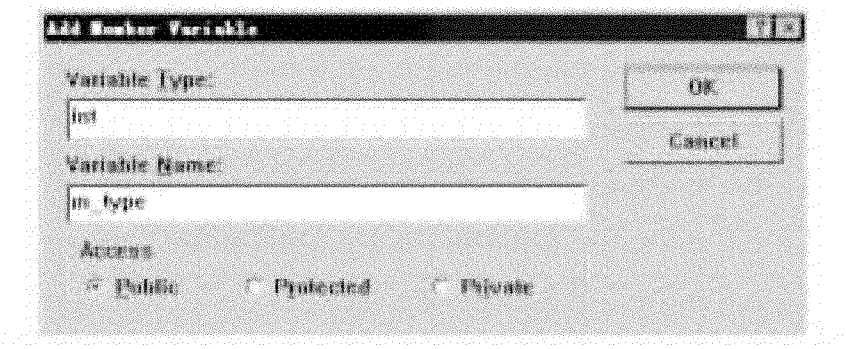



图6.11　“Add Member Variable”对话框

表6.3　在**CMyDrawView**类中添加的成员变量属性表

| 变量类型 | 变量名称 | 用途 |
|---|---|---|
| int | m_type | 当前选择图形类型 |
| COLORREF | m_color | 当前画笔颜色 |
| int | m_nmx | 窗口横坐标最大值 |
| int | m_nmy | 窗口纵坐标最大值 |
| int | m_width | 当前画笔宽度 |
| CBitmap | m_pbmp | 位图GDI对象 |
| CDC | m_pmdc | 存于内存中的设备环境变量 |
| CPoint | m_pold | 光标原位置 |
| CPoint | m_pnew | 光标新位置 |
| BOOL | m_bdoing | 标识当前是否在绘图 |

特别说明：

最简便的方法是直接打开MyHelloView.h文件，修改CMyDrawView类的定义：

```
class CMyDrawView : public CView
{
protected: // create from serialization only
    CMyDrawView();
    DECLARE_DYNCREATE(CMyDrawView)

// Attributes
public:
    CMyDrawDoc* GetDocument();
……
// Implementation
public:
    CPoint m_pold;
    CPoint m_pnew;
    BOOL m_bdoing;
    COLORREF m_color;
    int m_width;
    int m_type;
    CBitmap* m_pbmp;
    CDC* m_pmdc;
    int m_nmy;
    int m_nmx;

    virtual ~CMyDrawView();
    ……
    DECLARE_MESSAGE_MAP()
};
```

(2) 在构造函数中,初始化数据成员:

```
CMyDrawView::CMyDrawView()
{
   // TODO: add construction code here
   m_bdoing = FALSE;
   m_type = 0;
   m_width = 1;
   m_color = RGB(0,0,255);
   m_pmdc = new CDC;
   m_pbmp = new CBitmap;
}
```

### 6.5.4 为工具栏按钮编写代码

下面以工具栏上“颜色”按钮为例,说明为其添加响应函数和编写代码的方法,其他按钮完全类似。

**1.** 为“颜色”按钮的 **COMMAND** 消息添加响应函数

操作步骤如下:

(1) 选择“View”菜单的“ClassWizard”菜单项(或按 Ctrl + W 键),弹出“MFC ClassWizard”对话

框。

（2）选择对话框中的“Message Maps”标签，并进行如下设置：

```
Class name:CMyDrawView
Object IDs:ID_COLOR
Message:COMMAND
```

（3）单击“Add Function”按钮来增加新函数，在弹出的“Add Member Function”对话框中，保留默认函数名 OnColor，单击“OK”按钮，在 CMyDrawView 类中为工具栏上“颜色”按钮添加消息响应函数，如图 6.12 所示。

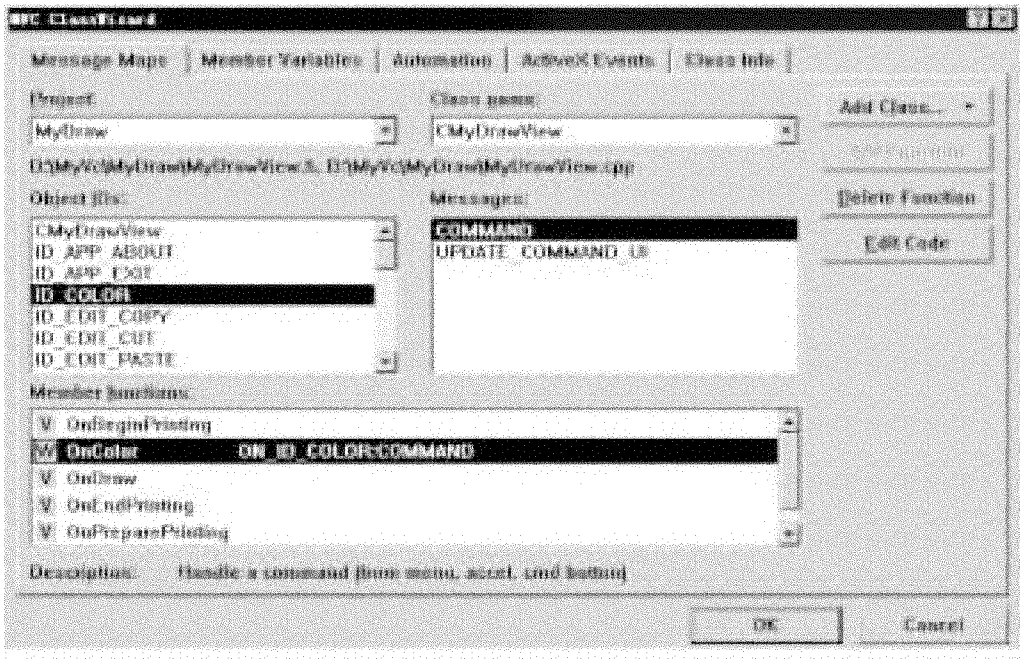



图 6.12　为“ID_COLOR”按钮添加消息响应函数

**2.** 编写程序代码

在图 6.12 中，单击“Edit Code”按钮（或打开 MyDrawView.cpp 文件，定位于 OnColor()函数），在消息响应函数 OnColor()中编写如下代码：

```
void CMyDrawView::OnColor()
{
    CColorDialog dlg; //构造颜色通用对话框类 CColorDialog 的对象 dlg
    dlg.m_cc.Flags |= CC_PREVENTFULLOPEN | CC_RGBINIT;
    dlg.m_cc.rgbResult = m_color;
    if(dlg.DoModal() == IDOK)
    {
        m_color = dlg.GetColor(); //将获取选定的颜色赋给当前画笔颜色变量
    }
}
```

**3.** 代码分析

在函数 OnColor()中，语句：

```
dlg.m_cc.rgbResult = m_color;
```

将当前绘画颜色 m_color 作为启动颜色对话框的默认颜色，因为变量 m_color 存储当前绘画颜色值。

语句：

```
m_color = dlg.GetColor();
```

获取颜色对话框中选中的颜色，并赋予 m_color 作为绘画颜色值。

**4.** 处理其他按钮

（1）完全类似地，为工具栏上其他按钮分别创建 COMMAND 消息响应函数，如表 6.4 所示。

表 **6.4** 工具栏上诸按钮消息响应函数

| Object IDs | Messages | Member Functions |
|---|---|---|
| ID_COLOR | COMMAND | OnColor |
| ID_FILL | COMMAND | OnFill |
| ID_LINE | COMMAND | OnLine |
| ID_RECT | COMMAND | OnRect |
| ID_ELLIRECT | COMMAND | OnEllirect |
| ID_ELLIPSE | COMMAND | OnEllipse |
| ID_WIDTH1 | COMMAND | OnWidth1 |
| ID_WIDTH2 | COMMAND | OnWidth2 |
| ID_WIDTH3 | COMMAND | OnWidth3 |

（2）编写代码

```
void CMyDrawView::OnEllipse()
{
   m_type = 4;
}

void CMyDrawView::OnEllirect()
{
   m_type = 3;
}

void CMyDrawView::OnFill()
{
   m_type = 0;
}

void CMyDrawView::OnLine()
```

```
{
    m_type = 1;
}

void CMyDrawView::OnRect()
{
    m_type = 2;
}

void CMyDrawView::OnWidth1()
{
    m_width = 1; // TODO: Add your command handler code here
}

void CMyDrawView::OnWidth2()
{
    m_width = 2;// TODO: Add your command handler code here
}

void CMyDrawView::OnWidth3()
{
    m_width = 3;// TODO: Add your command handler code here

}
```

### 6.5.5　编辑光标资源

为了使操作直观、形象，用鼠标光标来标识当前选择的绘图类型，即画直线时，光标为直线；画圆时，光标为圆；填充区域时，光标类似“填充区域”按钮等的图形形状。

编辑光标资源的步骤如下：

（1）在“Workspace”窗口中的“ResourceView”标签中，右击“MyDraw resource”选项，在弹出的快捷菜单中，选择“Insert”选项，VC++6.0显示“Insert Resource”对话框，如图6.13所示。

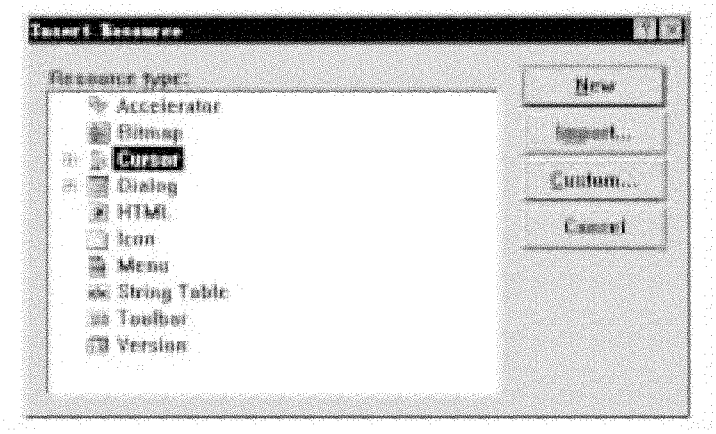



图6.13　“Insert Resource”对话框

（2）选择“Cursor”选项，然后单击“New”按钮。VC++6.0显示光标资源编辑器，打开“Graphics”工具箱，编辑所需要的光标图案，如图6.14所示。

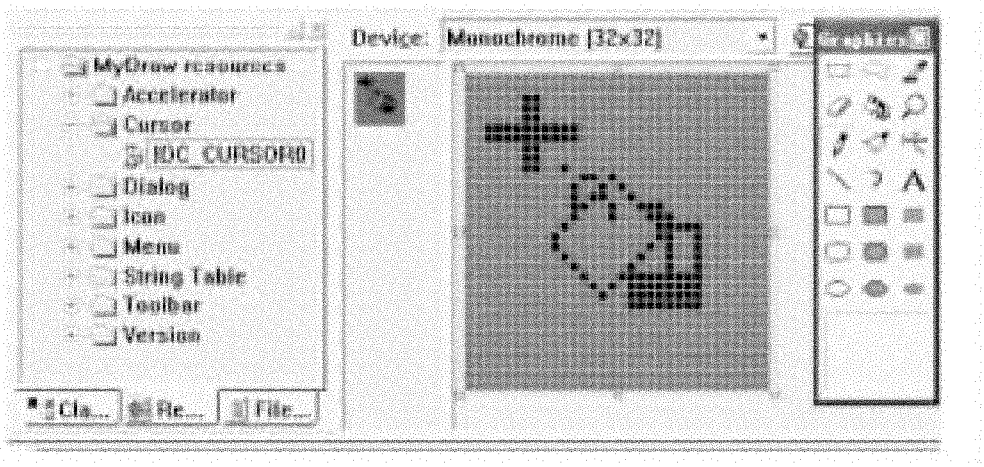



图6.14 编辑好的“填充区域”光标资源IDC_CURSOR0

（3）用完全类似的方法编辑其余光标资源：画直线（IDC_CURSOR1）、画矩形（IDC_CURSOR2）、画椭圆（IDC_CURSOR3）和画圆（IDC_CURSOR4），如图6.15所示。

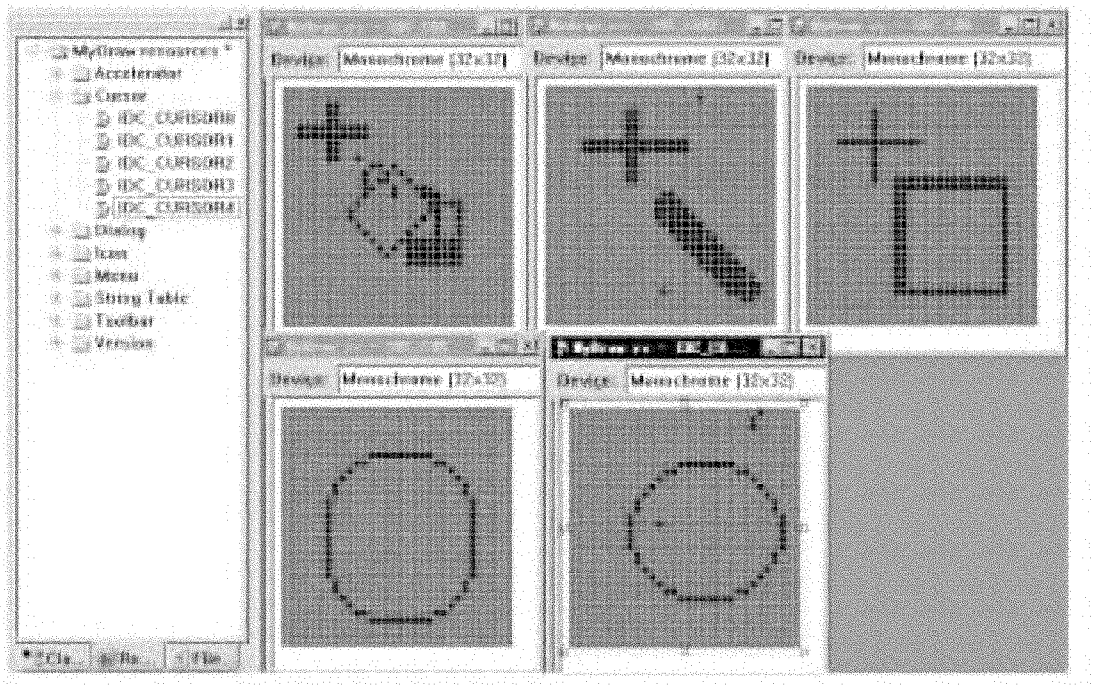


图6.15 编辑好的光标资源

### 6.5.6 编写响应鼠标消息WM_SETCURSOR的代码

当鼠标引起光标移动时，向窗口发送WM_SETCURSOR消息，因此可以在响应函数中改变鼠标的形状，本程序中使用了自己定义的光标，从而标识当前选择的绘图类型。

添加鼠标消息 WM _ SETCURSOR 的处理函数，操作步骤如下：

（1）选择“View”菜单的“ClassWizard”菜单项，弹出“MFC ClassWizard”对话框。

（2）在“MFC ClassWizard”对话框的“Message Maps”标签中，并进行如下设置：

```
Class name: CMyDrawView
Object IDs: CMyDrawView
Message: WM _ SETCURSOR
```

（3）单击“Add Function”按钮就在 CMyDrawView 类中添加了 WM _ SETCURSOR 消息的响应函数 OnSetCursor，如图 6.16 所示。

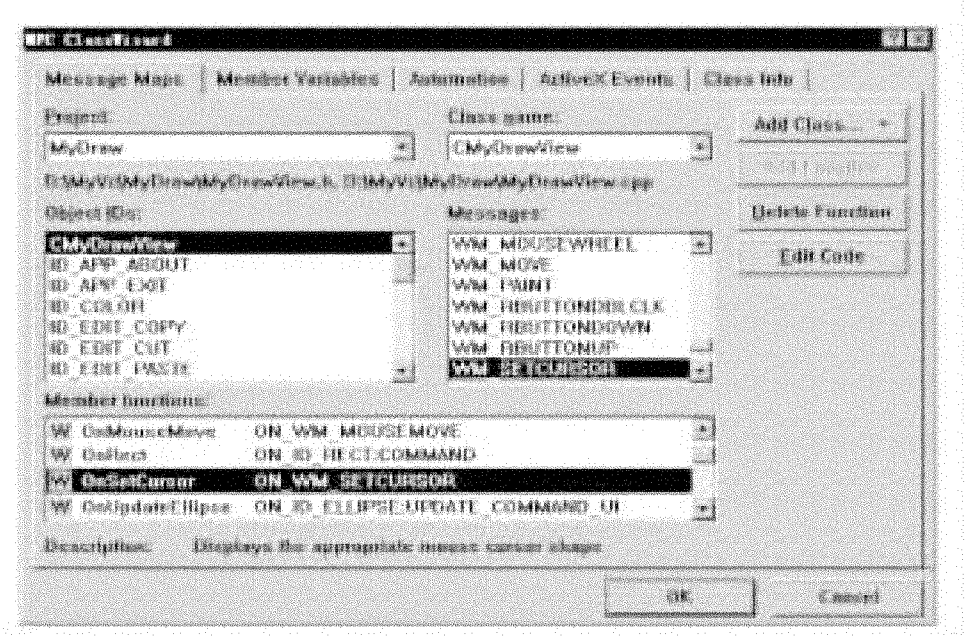



图 6.16　添加 WM _ SETCURSOR 消息响应函数

（4）单击“Edit Code”按钮，定位于 OnSetCursor( )函数，编写代码：

```
BOOL CMyDrawView::OnSetCursor(CWnd * pWnd, UINT nHitTest, UINT message)
{
   // TODO: Add your message handler code here and/or call default
   switch(m _ type)
   {
   case 0:
        ::SetCursor(AfxGetApp( ) -> LoadCursor(IDC _ CURSOR0));
        break;
   case 1:
        ::SetCursor(AfxGetApp( ) -> LoadCursor(IDC _ CURSOR1));
        break;
   case 2:
        ::SetCursor(AfxGetApp( ) -> LoadCursor(IDC _ CURSOR2));
        break;
   case 3:
        ::SetCursor(AfxGetApp( ) -> LoadCursor(IDC _ CURSOR3));
```

```
            break;
        case 4:
            ::SetCursor(AfxGetApp()->LoadCursor(IDC_CURSOR4));
            break;
        default:
            CView::OnSetCursor(pWnd, nHitTest, message);
            //::SetCursor(AfxGetApp()->LoadCursor(IDC_ARROW));
            break;
        }
        return TRUE;
    // return CView::OnSetCursor(pWnd, nHitTest, message);
    }
```

函数中根据当前选择的操作类型(填充图形、画线、画矩形、画圆角矩形或画椭圆)的标识 m_type 的值,确定选用的鼠标光标类型。

### 6.5.7 编写响应鼠标动作代码

本例使用鼠标画图,需要处理 3 个鼠标动作:

- WM_LBUTTONDOWN(左键按下)
- WM_MOUSEMOVE(鼠标移动)
- WM_LBUTTONUP(左键松开)

**1.** 添加消息响应函数

完全类似于前一小节 6.5.6 的方法,用 MFC ClassWizard 自动生存 3 个消息响应函数,它们是 OnLButtonDown()、OnMouseMove()和 OnLButtonUp()。

**2.** 编写程序代码

由于不需要基类的鼠标消息处理,所以将自动生存的调用基类的鼠标消息处理函数注释掉。

(1) OnLButtonDown()函数的代码如下:

```
void CMyDrawView::OnLButtonDown(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if(m_bdoing) return;          //如果正在画图,则返回
    SetCapture();                 //捕获鼠标信息
    m_bdoing = true;              //画图标识
    m_pnew = point;               //将当前鼠标点赋给画图的起点 m_pnew
    m_pold = point;               //将当前鼠标点赋给临时点 m_pold
    if(m_type == 0)               //如果是填充,则做如下处理
    {
        CBrush * poldbrush;       //定义保存内存 DC 中原画刷的指针
        CBitmap * poldbmp;
        CBrush bfill;
        bfill.CreateSolidBrush(m_color);
        poldbrush = m_pmdc->SelectObject(&bfill);
```

```
        poldbmp = m_pmdc->SelectObject(m_pbmp);
        m_pmdc->ExtFloodFill(point.x,point.y,m_pmdc->GetPixel(point),
                    FLOODFILLSURFACE);
        Invalidate(FALSE);        //使窗口无效,重画视窗
        m_pmdc->SelectObject(poldbrush);
        m_pmdc->SelectObject(poldbmp);
        m_bdoing = FALSE;         //画图结束
    }
//  CView::OnLButtonDown(nFlags, point);
}
```

（2）OnMouseMove()函数的代码如下：

```
void CMyDrawView::OnMouseMove(UINT nFlags, CPoint point)
{
    if(m_bdoing)                  //如果正在画图,执行如下代码
    {
        CDC* pDC = GetDC();       //获得显示 DC 的指针
        CBitmap* poldbmp = m_pmdc->SelectObject(m_pbmp);
        CPen pen;
        pen.CreatePen(PS_SOLID,m_width,m_color);
        CPen* poldpen = pDC->SelectObject(&pen);
        CBrush* poldbrush = (CBrush*)pDC->SelectStockObject(NULL_BRUSH);
        CRect rectold(m_pold,m_pnew);
        rectold.NormalizeRect();
        rectold.InflateRect(m_width,m_width);
        pDC->BitBlt(rectold.left,rectold.top,rectold.Width(),
            rectold.Height(),m_pmdc,rectold.left,rectold.top,SRCCOPY);
        CRect rectnew(m_pold,point);
        switch(m_type)
        {
        case 1:
             pDC->MoveTo(m_pold);
             pDC->LineTo(point);
             break;
        case 2:
             pDC->Rectangle(rectnew);
             break;
        case 3:
             pDC->RoundRect(m_pold.x,m_pold.y,point.x,point.y,10,10);
             break;
        case 4:
             pDC->Ellipse(rectnew);
             break;
        default:
            break;
        }
        m_pmdc->SelectObject(poldbmp);
```

```
        pDC->SelectObject(poldpen);
        pDC->SelectObject(poldbrush);
        ReleaseDC(pDC);
        m_pnew=point;
    }
//    CView::OnMouseMove(nFlags, point);
}
```

(3) OnLButtonUp()函数的代码如下:

```
void CMyDrawView::OnLButtonUp(UINT nFlags, CPoint point)
{
    // TODO: Add your message handler code here and/or call default
    if(m_bdoing)
    {
        m_bdoing=FALSE;
        CBitmap* poldbmp=m_pmdc->SelectObject(m_pbmp);
        CPen pen;
        pen.CreatePen(PS_SOLID,m_width,m_color);
        CPen* poldpen=m_pmdc->SelectObject(&pen);
        CBrush* poldbrush=(CBrush*)m_pmdc->SelectStockObject(NULL_BRUSH);
        CRect rect(m_pnew,m_pold);
        switch(m_type)
        {
        case 1:
            m_pmdc->MoveTo(m_pold);
            m_pmdc->LineTo(point);
            break;
        case 2:
            m_pmdc->Rectangle(rect);
            break;
        case 3:
            m_pmdc->RoundRect(m_pnew.x,m_pnew.y,m_pold.x,m_pold.y,10,10);
            break;
        case 4:
            m_pmdc->Ellipse(rect);
            break;
        default:
            break;
        }
        Invalidate(FALSE);
        m_pmdc->SelectObject(poldbmp);
        m_pmdc->SelectObject(poldpen);
        m_pmdc->SelectObject(poldbrush);
    }
    ReleaseCapture();

    //CView::OnLButtonUp(nFlags, point);
}
```

**3.** 查看结果

(1) 编译、连接执行 MyDraw 程序，出现如图 6.6 所示的界面。

(2) 先选定作图类型，按下鼠标左键不放，拖动鼠标，可以看到屏幕上随鼠标移动画出的图形。

(3) 松开鼠标左键，作图完成。

(4) 拉动窗口，所画的图形消失。那么，怎样才能保留图形呢？在 6.6 节将解决这问题。

### 6.5.8　修改 OnDraw()函数

当松开鼠标时，首先将图形画在兼容位图上，然后调用 Invalidate()函数，重画窗口，调用 OnDraw()函数，将画在内存 DC 的图形拷贝至屏幕。此时的 OnPraw()函数的代码为：

```
void CMyDrawView::OnDraw(CDC * pDC)
{
   CMyDrawDoc * pDoc = GetDocument();
   ASSERT_VALID(pDoc);
   // TODO: add draw code for native data here
   CBitmap * poldbmp = m_pmdc->SelectObject(m_pbmp);
   pDC->BitBlt(0,0,m_nmx,m_nmy,m_pmdc,0,0,SRCCOPY);
   m_pmdc->SelectObject(poldbmp);
}
```

### 6.5.9　技术要点

关于 **Invalidate()**函数：

Invalidate()函数是 CWnd 类的成员函数，它的功能是使整个客户区无效，引起窗口重画，调用视图类的 OnDraw()函数。Invalidate()函数的调用方式如下：

```
Invalidate(FALSE);
```

## 6.6　完善绘图程序

下面为 MyDraw 程序增加菜单和快捷键选择方式，使菜单和工具栏按钮显示更新状态，改善人机交互功能，完善绘图程序。

要使某一菜单项与工具栏上的按钮共享同一消息响应函数，也就是完成同一操作，只要它们的 ID 标识符相同即可。所以，下面的工作是：编辑菜单资源，设置与工具栏上按钮相应的同一 ID；添加键盘加速键；菜单项的状态更新。

### 6.6.1　编辑菜单资源

通过菜单资源编辑器设计菜单，操作步骤如下：

**1.** 打开菜单资源编辑器

在“Workspace”窗口的资源(ResourceView)列表中,双击“Menu”下的“IDR _ MAINFRAME”项,打开菜单资源编辑器,可以看到应用程序默认定义的菜单资源形式,如图 6.17 所示。

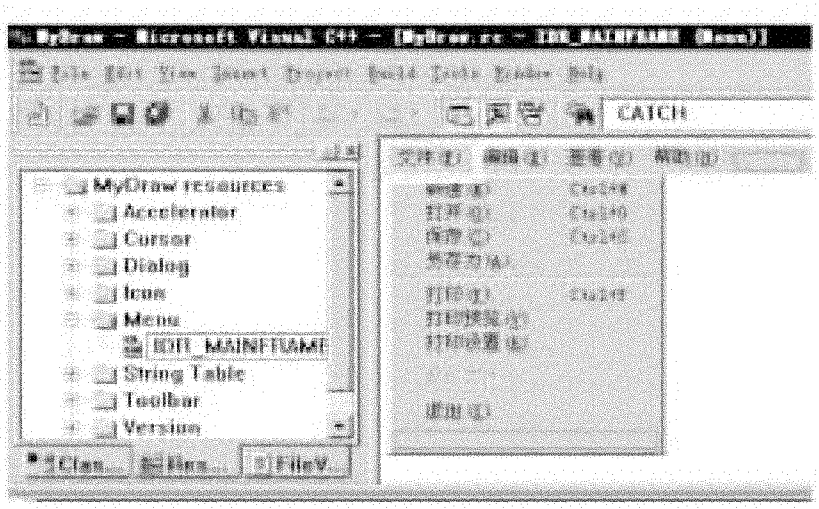



图 6.17 默认菜单资源形式

**2.** 增加菜单项

在图 6.17 的“帮助”菜单右侧有一个空菜单项,将其拖动至“查看”与“帮助”菜单项之间,双击该菜单项,在弹出的“Menu Item Properties”对话框中,按图 6.18 所示增加一个“绘画”菜单项。最后单击右上角按钮关闭对话框。

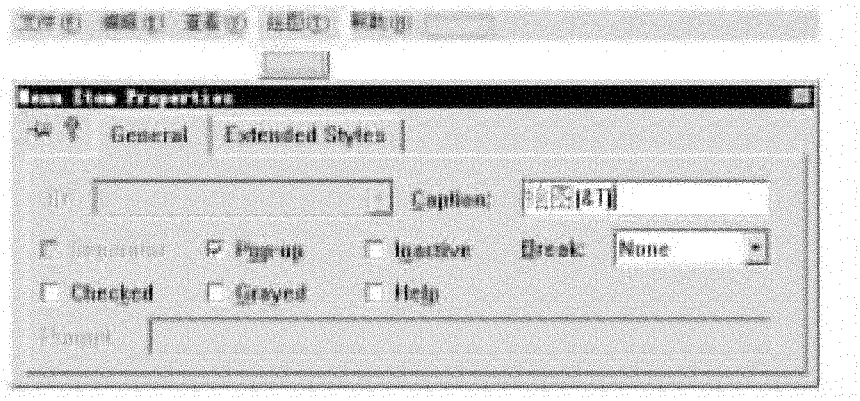



图 6.18 添加“绘图”菜单项

**3.** 增加菜单选项

(1) 双击“绘图”菜单项下的空菜单项,弹出的“Menu Item Properties”对话框,增加一个“直线”菜单项,在“ID”下拉式列表框中选择“ID _ LINE”(工具栏上“直线”按钮的标识符)或直接输入“ID _ LINE”,如图 6.19 所示。最后单击右上角关闭对话框。

(2) 完全类似地,在“绘图”菜单下添加表 6.5 所列的菜单项。

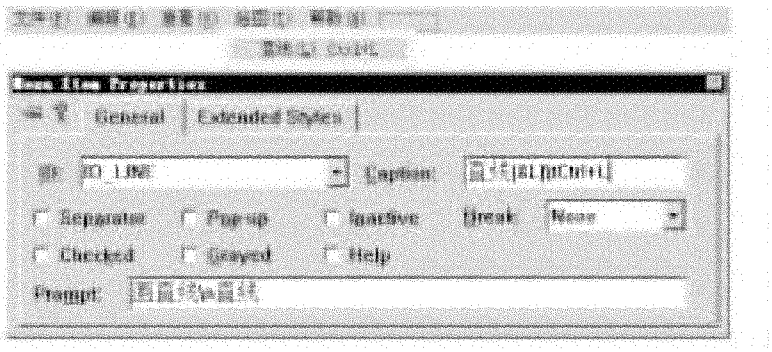



图 6.19　“直线”菜单项的“Menu Item Properties”对话框

表 **6.5**　“绘图”菜单中新添菜单项的属性设置

| 菜单项 | ID | Caption | Prompt(自动显示) |
|---|---|---|---|
| 矩形 | ID _ RECT | 矩形[&R]\tCtrl + R | 画矩形\n矩形 |
| 圆角矩形 | ID _ ELLIRECT | 圆角矩形[&I]\tCtrl + I | 画圆角矩形\n圆角矩形 |
| 椭圆 | ID _ ELLIPSE | 椭圆[&E]\tCtrl + E | 画椭圆\n椭圆 |

(3) 在“绘图”的弹出菜单中继续加入分隔符和表 6.6 所列的 3 个菜单。

表 **6.6**　“绘图”菜单的另 **3** 个菜单项的属性设置

| 菜单项 | ID | Caption | Prompt |
|---|---|---|---|
| 线宽 | | 线宽[&W]\tCtrl + W | 线宽 |
| 颜色 | ID _ COLOR | 颜色[&C]\tCtrl + C | 选择要使用的颜色\n选颜色 |
| 填充 | ID _ FILL | 填充[&F]\tCtrl + F | 区域填充\n填充 |

其中“线宽”一项除了设置 Caption 外，还要将“Pop-up”复选框选中。

(4) 在“线宽”的弹出菜单中加入表 6.7 所列的 3 个子菜单。编辑后的菜单资源，如图 6.20 所示。

表 **6.7**　“线宽”弹出菜单中的 **3** 个子菜单项的属性设置

| 菜单项 | ID | Caption | Prompt |
|---|---|---|---|
| 线宽一 | ID _ WIDTH1 | 1 | 线宽为一个像素\n线宽为一 |
| 线宽二 | ID _ WIDTH2 | 2 | 线宽为二个像素\n线宽为二 |
| 线宽三 | ID _ WIDTH3 | 3 | 线宽为三个像素\n线宽为三 |

### 6.6.2　添加键盘加速键

前面用“&”引入了菜单选项的快捷键，下面通过定义加速键方式来为程序增加快捷键，两者在操作上是有区别的。

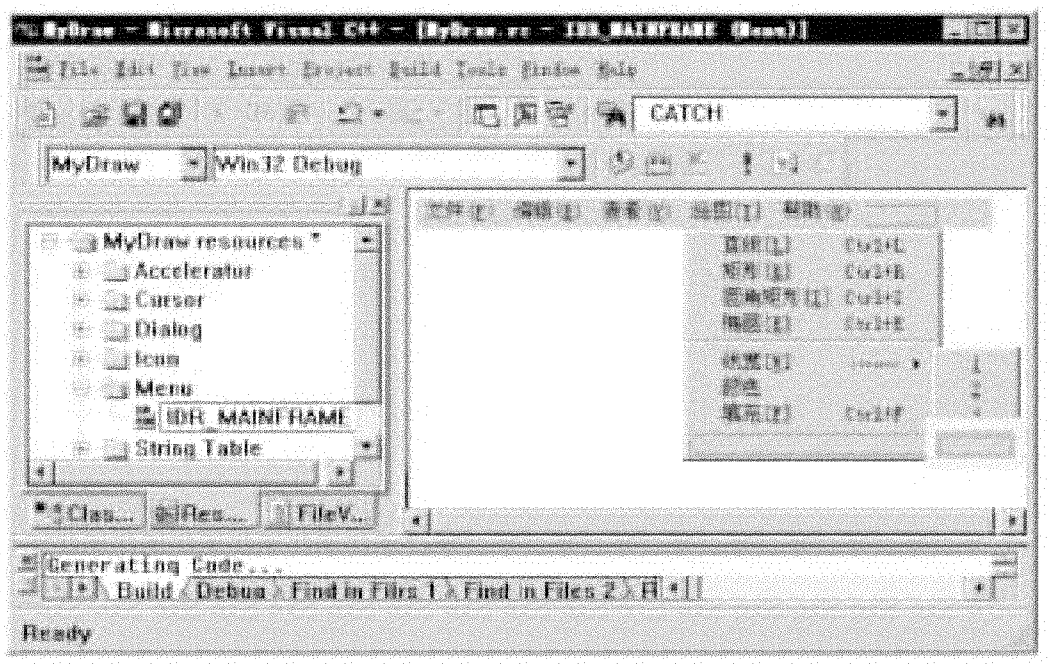



图 6.20 编辑后的菜单资源

**1.** 添加选择“直线”的快捷键 **Ctrl + L**

（1）打开“Workspace”窗口，选择“Resource View”标签。

（2）展开 MyDraw Resources，双击“Accelerator”下的“IDR _ MAINFRAME”项，打开快捷键资源编辑器，如图 6.21 所示。

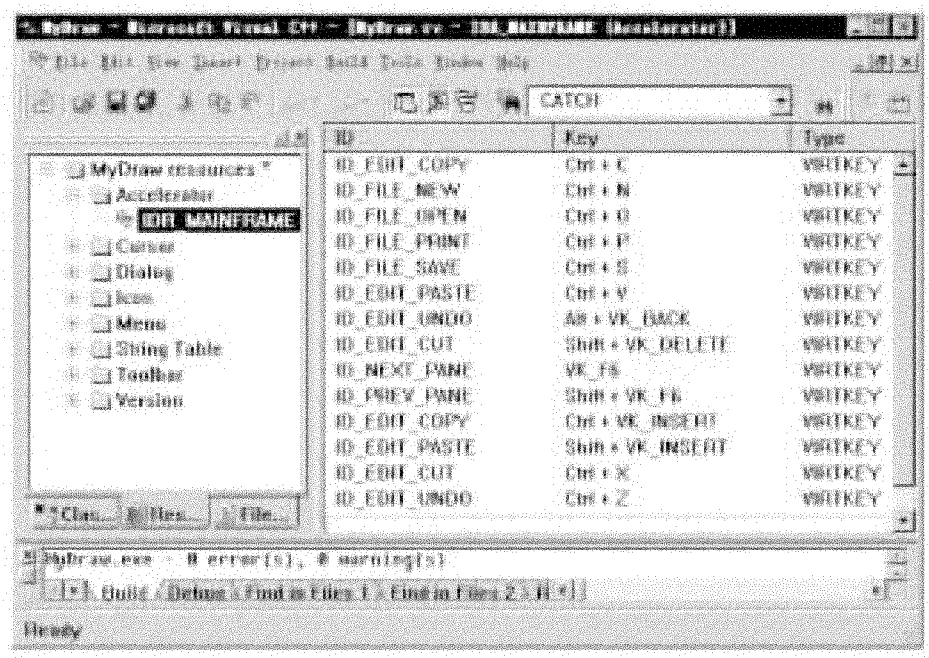



图 6.21 快捷键资源

(3) 在最后的虚框上双击鼠标左键(或者单击鼠标右键并在弹出的菜单中选择“New Accelerator”菜单项),弹出“Accel Properties”对话框,如图6.22所示。对框中各项进行如下设置:

```
ID: ID_LINE
Key: L
Modifiers: Ctrl
Type: VirtKey
```

当输入焦点从“ID”列表框转至“Key”组合框时,资源编辑器自动在“ID”下拉式列表框中的“ID_LINE”后加上了“=32773”,这是在加入菜单项时给该ID定义的整数值。

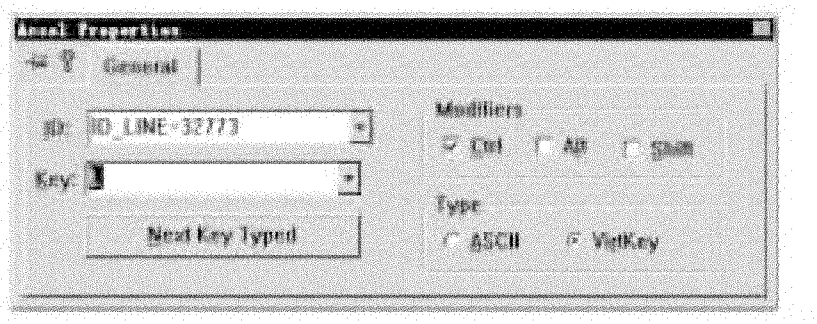



图6.22　“Accel Properties”对话框

这样就为应用程序添加了选择“直线”的快捷键Ctrl+L。

**2.** 给其余菜单项添加快捷键

这些工作与给“直线”菜单项添加快捷键完全类似,留给读者完成。

### 6.6.3　菜单项的状态更新

有时,需要禁止某个菜单项,使之成为非活动状态,不允许用户选择。有的菜单项就是以布尔值的方式出现的,选中一次,值为真,同时旁边有一选中标志;再选一次,值为假,标志消失。

菜单项的状态更新控制有两种方式:

- SetCheck方式　该方式在菜单命令左边作复选标记(画√)。
- Enabele/Disable方式　该方式使菜单命令禁止(呈灰色)或恢复。

**1.** 为“直线”菜单项作复选标记状态更新

(1) 从“View”菜单项中选择“ClassWizard”。

(2) 在“MFC ClassWizard”对话框中,选择“Message Maps”标签,进行如下设置:

```
Class name: CMyDrawView
Object IDs: ID_LINE
Messages: UPDATE_COMMAND_UI
```

(3) 单击“Add Function”按钮,在弹出的“Add Member Function”对话框中,保留默认函数名OnUpdateLine,单击“OK”按钮,在CMyDrawView类中为“直线”菜单项(也就是工具栏上“直线”按钮)状态更新添加消息响应函数。

(4) 单击“Edit Code”按钮，定位于 OnUpdateLine( )函数，并添加菜单命令左边作复选标记代码：

```
void CMyDrawView::OnUpdateLine(CCmdUI * pCmdUI)
{    pCmdUI->SetCheck(m_type==1);
}
```

**2.** 代码分析

“UPDATE_COMMAND_UI”事件在“绘图”菜单显示之前发生。在本例程中，“绘图”菜单包括 7 个菜单项：“直线”、“ 矩形”、“ 圆角矩形”、“ 椭圆”、“线宽”、“颜色”和“填充”，当用户打开“绘图”菜单时，它们就会显示出来。在打开菜单之前，会发生 UPDATE_COMMAND_UI 事件，此时与菜单项相连的代码就会被执行。

如果用户选取了工具栏或“绘图”菜单中的“直线”，所以就执行了其消息响应函数：

```
void CMyDrawView::OnLine()
{
   m_type=1;
}
```

当再次打开“绘图”菜单时，UPDATE_COMMAND_UI 事件会发生。需要在此事件函数中添加语句：

```
pCmdUI->SetCheck(m_type==1);
```

用来根据变量的内容在相应的菜单处加一个选中标记。

**3.** 为其他菜单项作复选标记状态更新

完全类似地，可为其他菜单项(也就是工具栏上其他按钮)分别创建如表 6.8 所示的 UPDATE_COMMAND_UI 消息响应函数，并编写复选标记状态更新代码，这项工作留给读者完成。

表 **6.8** 工具栏上诸按钮消息响应函数

| Object IDs | Messages | Member Functions |
|---|---|---|
| ID_FILL | UPDATE_COMMAND_UI | OnUpdateFill |
| ID_LINE | UPDATE_COMMAND_UI | OnUpdateLine |
| ID_RECT | UPDATE_COMMAND_UI | OnUpdateRect |
| ID_ELLIRECT | UPDATE_COMMAND_UI | OnUpdateEllirect |
| ID_ELLIPSE | UPDATE_COMMAND_UI | OnUpdateEllipse |
| ID_WIDTH1 | UPDATE_COMMAND_UI | OnUpdateWidth1 |
| ID_WIDTH2 | UPDATE_COMMAND_UI | OnUpdateWidth2 |
| ID_WIDTH3 | UPDATE_COMMAND_UI | OnUpdateWidth3 |

# 6.7　快捷菜单

几乎在所有的 Windows 应用程序中，单击鼠标右键会弹出快捷菜单，以便用户高效地选择常用功能。快捷菜单也称为弹出式菜单。

为 MyDraw 绘图程序添加选择绘图类型（直线、矩形、椭圆等）和线型宽度快捷菜单。如图 6.23 所示，在绘画区单击鼠标右键，弹出快捷菜单供用户选择绘图类型和线型宽度。选择其中的某菜单项后，快捷菜单自动消失，回到程序主窗口，再进行绘图操作时，按选择的类型进行绘图。

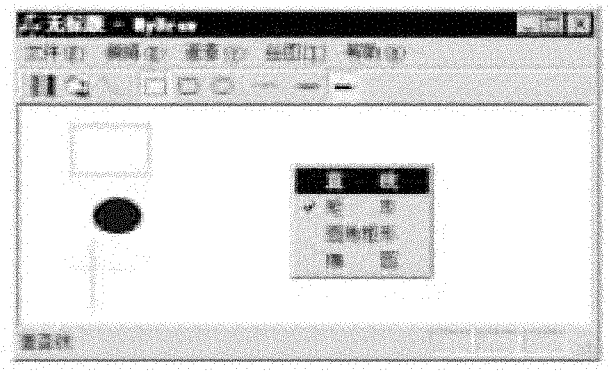


图 6.23　弹出快捷菜单

## 6.7.1　编辑快捷菜单资源

（1）打开“Workspace”窗口，选择“Resource View”标签。

（2）展开 MyDraw Resources，在“Menu”项上单击鼠标右键，弹出快捷菜单，选择“Insert Menu”项，增加一个空菜单，默认 ID 号为“IDR _ MENU1”。

（3）在“IDR _ MENU1”上单击鼠标右键，弹出快捷菜单，选择“Properties”项，将其 ID 号改为“IDR _ POPUP _ MENU”。

（4）用鼠标左键双击左上角的空白菜单，弹出“Menu Item Properties”对话框，选中“General”标签中的“Pop-up”，并在“Caption”中输入“Popup1”文本，该菜单下面自动出现空白菜单项。

（5）用常用添加菜单的方法加入表 6.9 所列菜单项，编辑好的快捷菜单如图 6.24 所示。

表 **6.9**　快捷菜单资源属性表

| ID | Caption | 说明 |
|---|---|---|
| ID _ LINE | 直线 | 绘制直线 |
| ID _ RECT | 矩形 | 绘制矩形 |
| ID _ ELLIPRECT | 圆角矩形 | 绘制圆角矩形 |
| ID _ ELLIPSE | 椭圆 | 绘制椭圆 |

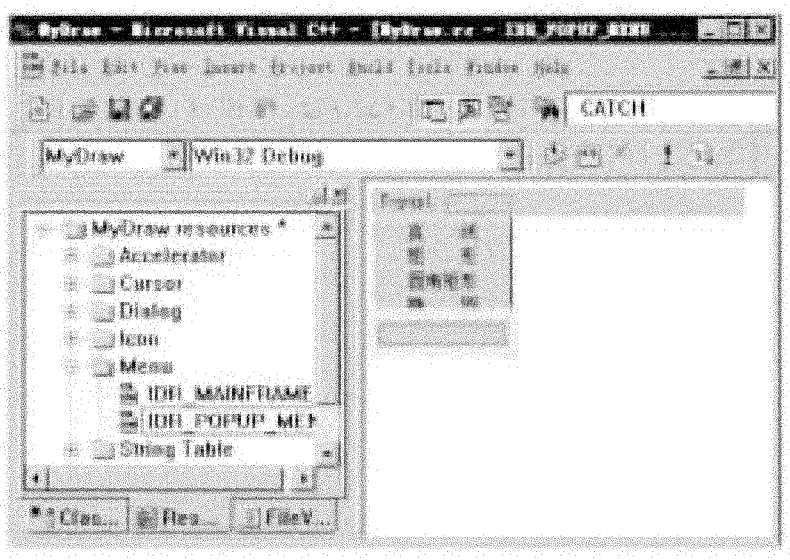



图 6.24 加入菜单资源

### 6.7.2 建立快捷菜单与 CMainFrame 类的关联

要显示快捷菜单，应用程序必须在视图类或主框架类中处理 WM _ CONTEXTMENU 消息。所以必须将快捷菜单与其类相关联。我们来建立与主框架类的关联，也就是对快捷菜单的操作，纳入到主框架类的处理中，操作步骤如下：

（1）选中“Resource View”中的“IDR _ POPUP _ MENU”。

（2）选择“View”菜单下的“ClassVizard”菜单，打开“MFC ClsssWizard”对话框，由于新加入了菜单资源“IDR _ POPUP _ MENU”，类向导提示为其创建一个类，或为其选择一个已经存在的类。选择已经存在的类，如图 6.25 所示。

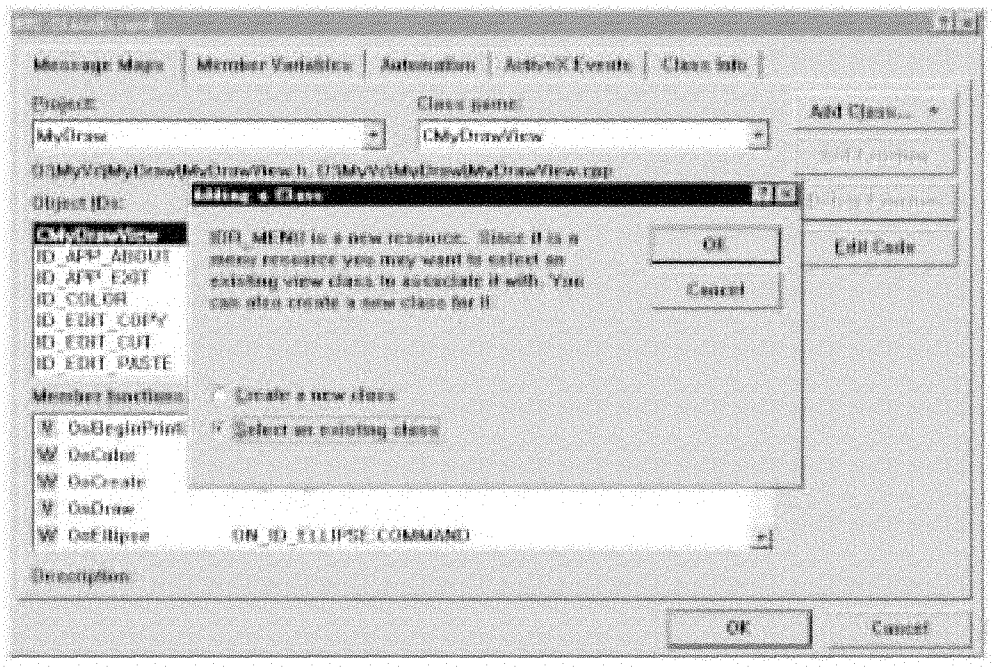



图 6.25 为菜单资源选择一个类

（3）单击“OK”按钮，弹出一个列有已经存在的类列表对话框，选择主框架类 CMainFrame，如图 6.26 所示。单击“Select”按钮，返回“MFC ClassWizard”类向导。

（4）单击“OK”按钮，就将菜单资源“IDR _ POPUP _ MENU”与主框架类 CMainFrame 建立了关联。

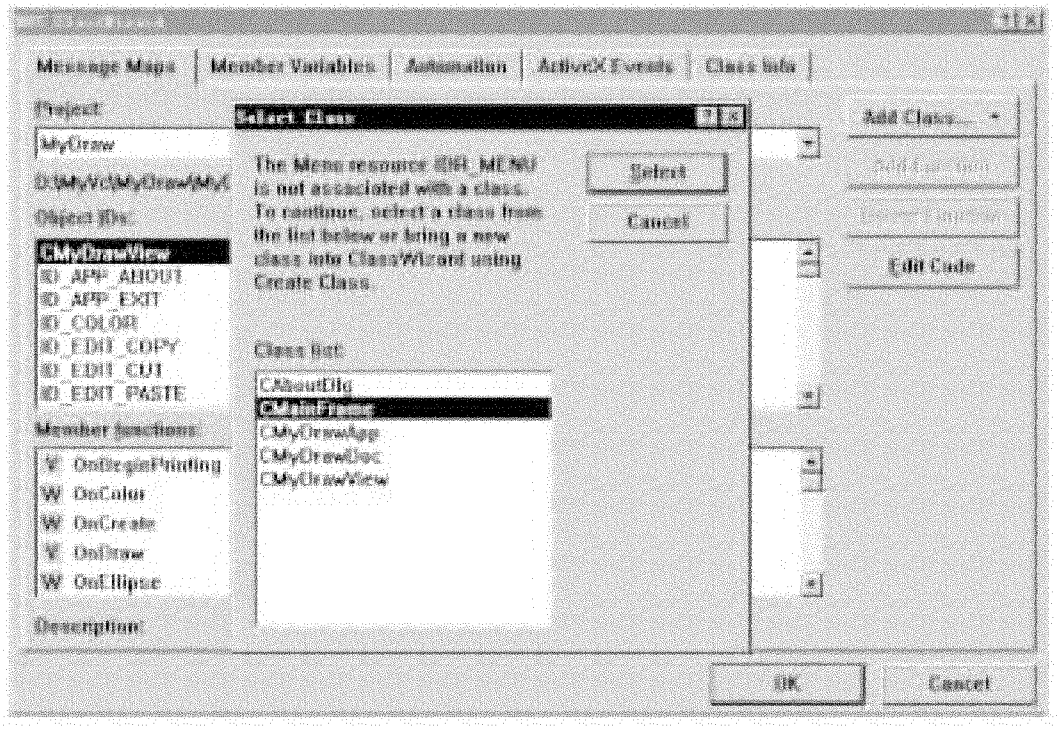



图 6.26　为菜单资源选择 CMainFrame 类

### 6.7.3　显示快捷菜单

快捷菜单资源“IDR _ POPUP _ MENU”已与主框架类 CMainFrame 建立了关联，所以，显示快捷菜单，就是在主框加类中处理 WM _ CONTEXTMENU 消息。

**1.** 显示快捷菜单

操作步骤如下：

（1）在“MFC ClsssWizard”对话框中，选择“Message Maps”标签。并进行如下设置：

```
Project: MyDraw
Class name: CMainFrame
Object Ids: CMainFrame
Messages: WM _ CONTEXTMENU
```

（2）单击“Add Function”按钮，接受默认函数名 OnContextMenu，按“OK”按钮。

（3）单击“Edit Code”按钮，定位于 OnContextMenu 函数，在函数体中编写显示快捷菜单“IDR _ POPUP _ MENU”的代码：

```
void CMainFrame::OnContextMenu(CWnd * pWnd, CPoint point)
```

```
{
    // TODO: Add your message handler code here
    CMenu menu;
    if(menu.LoadMenu(IDR_POPUP_MENU))  //加载快捷菜单资源
    {
    CMenu *pPopup = menu.GetSubMenu(0);
    pPopup->TrackPopupMenu(TPM_LEFTALIGN |
                           TPM_RIGHTBUTTON,point.x,point.y,this);
    }
}
```

**2.** 代码分析

当用户单击鼠标右键时，该点的坐标被传送给函数 OnContextMenu()。LoadMenu()是 CMenu 类的成员函数，装入菜单资源，参数 IDR_POPUP_MENU 为菜单资源的 ID 号，GetSubMenu()也是 CMenu 类的成员函数，返回 CMenu 的指针，参数 0 为弹出式菜单在菜单资源中的索引(本例中，只定义了一个弹出式菜单 IDR_POPUP_MENU，因此索引为 0，如果定义了多个弹出式菜单，其索引分别 0，1，2，……等)。函数 TrackPopupMenu()在指定的位置(参数 point.x，point.y)显示弹出式菜单，并跟踪用户在弹出式菜单中选择的菜单项，第一个参数为弹出式菜单的显示风格和鼠标按钮风格。菜单的显示有 3 种选择，鼠标按钮有两种选择，分别如下：

- TPM_CENTERALIGN　弹出式菜单的水平中心为 point.x。
- TPM_LEFTALIGN　弹出式菜单的左边位于 point.x。
- TPM_RIGHTALIGN　弹出式菜单的右边位于 point.x。
- TPM_LEFTBUTTON　弹出式菜单跟踪鼠标左键。
- TPM_RIGHTBUTTON　弹出式菜单跟踪鼠标右键。

参数 this 为主框架窗口的指针，表示弹出式菜单属于主框架窗口。

**3.** 查看结果

(1) 编译、运行 MyDraw 程序，出现如图 6.6 所示的运行界面。

(2) 在窗口中单击鼠标右键，弹出快捷菜单，并在当前绘图类型(矩形)菜单项作了复选标记(画√)，如图 6.23 所示。

(3) 选择绘图类型，比如选“椭圆”，快捷菜单消失，工具栏上的“椭圆”按钮立即作了选中标记。

要特别提醒读者注意的是：由于快捷菜单上、工具栏按钮有相同的 ID 号，前面已编写了工具栏按钮的消息响应函数，所以它们共享同一消息响应函数。如果没有编写消息响应函数，则会发现单项均呈灰色为无效状态，这是因为还没有加入菜单的消息处理函数的像故。

## 6.8 技术要点

### 6.8.1 CPoint 类

CPoint 类是对 Windows 结构 POINT 的封装，凡是能用 POINT 结构的地方都可以用 CPoint 代

替。

结构 POINT 表示屏幕上的一个二维点，其定义为：

```
Typedef struct tagPOINT
{
LONG x;
LONG y;
}POINT;
```

**1. CPoint 类的构造函数**

CPoint 类有如下 5 个构造函数：

- CPoint( )
- CPoint( int x, int y)
- CPoint( POINT initPt)
- CPoint( SIZE initSize)
- CPoint( DWORD dwPoint)

**2. CPoint 类常用操作符**

CPoint 类提供了一些重载运算符，使 CPoint 的操作更加容易。如运算符“ + ”、“ - ”、“ += ”和“ -= ”用于两个 CPoint 对象或一个 CPoint 对象与一个 CSize 对象的加减运算，运算符“ == ”和“ != ”用于比较两个 CPoint 对象是否相等。

### 6.8.2　CRect 类

Windows 定义了一个描述矩形的结构 RECT：

```
Typedef struct tagRECT
{
   LONG  left;
   LONG  top;
   LONG  right;
   LONG  botton;
}RECT, * PRECT, NEAR * NPRECT, FAR * LPRECT;
```

其中 left、top 分别表示矩形左上角顶点的横坐标和纵坐标，right、bottom 分别表示矩形右下角顶点的横坐标和纵坐标。

MFC 将 RECT 结构封装在 CRect 类中，凡能用 RECT 结构的地方都可以用 CRect 类代替。由于 CRect 类提供了一些成员函数和重载运算符，使得 CRect 类的操作更加方便。

（1）CRect 类的构造函数

CRect 类有如下 6 个构造函数：

- CRect( )
- CRect( int l, int t, int r, int b)
- CRect( const RECT& srcRect)
- CRect( LPCRECT lpSrcRect)

- CRect(POINT point, SIZE size)
- CRect(POINT topLeft, POINT bottomRight)

(2) CRect 类常用成员函数和操作符

CRect 类提供了表 6.10 所示的成员函数和操作符。

表 **6.10** **CRect** 类的成员函数和操作符

| 成员函数 | 描述 |
|---|---|
| BottomRight( ) | 返回 CRect 对象右下方的点的坐标 |
| CenterPoint( ) | 返回 CRect 对象的中央点的坐标 |
| CopyRect( ) | 将源矩形的尺寸拷贝到 CRect 对象 |
| Height | 返回 CRect 对象的高度 |
| InflateRect( ) | 增加 Ctect 对象的高度和宽度 |
| IntersectRect( ) | 返回一个由两个矩形之间的交叉区域定义的 CRect 对象 |
| IsRectEmpty( ) | 确定矩形是否为空的(宽度和高度是否为零) |
| IsRectNull( ) | 确定 RECT 结构中 4 个参数是否全为零 |
| PtInRect( ) | 测试指定的点是否在 CRect 对象中 |
| SetRect( ) | 设定 CRect 对象的尺寸 |
| SetRectEmpty( ) | 将 CRect 对象的所有坐标设置为零,形成一个空矩形 |
| Size( ) | 返回对象的尺寸 |
| TopLeft( ) | 返回 CRect 对象左上角的点的坐标 |
| Width( ) | 计算 CRect 对象的宽度 |
| operator LPCRECT | 将 CRect 对象转换为 LPCRECT |
| operator LPRECT | 将 CRect 对象转换为 LPRECT |
| operator = | 将一个矩形的尺寸拷贝到 CRect 对象上 |
| operator == | 测定两个 CRect 对象的边框坐标,以了解它们是否相等 |
| Operator != | 测定两个 CRect 对象的边框坐标以了解它们是否不等 |

### 6.8.3 CMenu 类

前面介绍过通过 UI 消息响应函数实现菜单选项的操作。而实际上通过 CMenu 类可以访问菜单对象,而且程序员不需要专门创建 CMenu 对象,只要通过 CWnd 的 GetMenu( )成员函数,即可返回一个临时 CMenu 对象的指针。还可以通过 GetSubMenu( )函数得到指向快捷菜单的 CMenu 指针。

CMenu 类的成员函数 LoadMenu( )、SetMenu( )可完成加载和连接菜单资源编辑器所创建的菜单资源到主窗口,这样就可以实现应用程序运行中调入所需的定制的菜单。

另一些成员函数,如 AppendMenu( )、ModifyMenu( )、InsertMenu( )和 DeleteMenu( )等则可通过菜单选项的 ID 号为线索完成菜单选项的操作,因此正确地使用 CMenu 类可使得动态增加、修改、删除指定菜单选项变得轻而易举。

# 习　题　六

1. 使用图形设备接口对象的一般步骤是什么？
2. 如何实现多个菜单项使用同一个响应函数？
3. 完成6.6.3节为其他菜单项创建UPDATE_COMMAND_UI消息响应函数并编写复选标记状态更新代码。
4. 编写图6.5所示的字体显示程序VC09.exe。
5. 请为MyDraw程序添加键盘字符输入与显示功能。
6. 完善MyDraw程序，添加图像显示和存储功能。

# 第7章

# 多媒体技术

## 本章导读

多媒体技术是综合图形、文字、声音和视频图像等多种媒体手段，用于传递和表达计算机信息的技术。本章将带领读者亲自去尝试多媒体开发，在程序中增加令人羡慕的多媒体效果，从中体会多媒体编程的乐趣。从而使读者：

- 了解多媒体程序的开发方法
- 掌握 MCI(Media Control Interface)编程技术
- 掌握进程条(Progress)、滑动条(Slider)和单选钮(Radio)等控件的编程技术

## 7.1 媒体播放器

图 7.1 所示是一个正在播放媒体的播放器，可以播放波形音频(WAVE)、MIDI 音频和视频(AVI)三种媒体，其中进程条显示播放的进度，滑动条可以显示和控制播放进度。

图 7.1 正在播放 Dancing.avi 视频文件的媒体播放器

本章将按应用程序功能，结合知识点进行目标分解，按以下步骤完成该系统的开发：

(1) 对媒体的 MCI 控制方式，构造一个 MCI 控制类：MCIClass。

（2）利用 MCIClass 类编写一个基于对话框的播放器。

（3）添加进程条，修改播放器。

（4）添加滑动条，完善播放器。

## 7.2　MCI 编程技术

Windows MCI（Media Control Interface）是控制多媒体设备的高层命令接口，提供了与设备无关的控制多媒体设备的方法。MCI 可控制所有 Windows 能驱动的多媒体设备，包括 CD 音频（CD Audio）、数字视频、动画、数字化波形声音、MIDI 音序器、录像机和影碟等。

MCI 包括在 Windows 系统的 MMSYSTEM.DLL 动态连接库中，用以协调多媒体事件和 MCI 设备驱动程序之间的通信。一些 MCI 设备驱动程序，如影碟设备驱动程序，可以直接控制目标设备；而另一些 MCI 设备驱动程序，如 Wave 和 MIDI 设备驱动程序，通过 MMSYSTEM 中的函数间接控制目标设备；还有一些 MCI 设备驱动程序则提供了与其他 Windows 动态连接库的高层接口。

### 7.2.1　多媒体程序的开发方法

开发多媒体应用程序，根据开发目的大致可分三类：第一是简单地调用媒体文件；第二是编辑处理文件；第三是控制多媒体硬件设备。

本章的重点在第一类，即调用媒体文件。因为其最常应用，也最容易掌握。在程序中调用多媒体文件方法主要有：

- 使用 MCIWND 窗口类控制波形音频文件、音频 CD 和视频等等。
- 使用 MCI（多媒体控件接口）。
- 使用 API 函数。

此外，还可以使用 PlaySound()、SndPlaySound()、MessageBeep()等函数控制播放音频文件。

以上方法中，使用 API 函数需要程序员编写大量代码，所以，比较简单的方法是使用 MCIWND 窗口类或直接发送 MCI 命令。

### 7.2.2　MCI 设备类型

使用 MCI 的应用程序通过指定 MCI 的设备类型来区分 MCI 设备，设备类型说明了设备的物理类型。使用 MCI 控制方式，在应用程序设计中，只操作 MCI。然后 MCI 通过具体的设备驱动程序来控制多媒体设备。但是，必须在 Windows 系统中安装 MCI 设备驱动程序才能在 Windows 中使用 MCI 设备，表 7.1 列出了可能用到的 MCI 设备类型及常用的设备驱动程序。

MCI 设备可分为两类：

（1）简单设备　描述简单 MCI 设备只需指定 MCI 设备名，如描述 CD 音频指定 cdaudio。

（2）复合设备　描述复合 MCI 设备不仅要说明设备类型，而且还要指定一个设备元素或媒体元素。对大多数复合设备来说，其设备元素是一个源或目标数据文件，其元素名就是相应的文件名。复合 MCI 设备的例子有 waveaudio、sequencer 和 mmmovie 等。比如要描述数字化波形声音，

不仅要指定其设备名 waveaudio，还要说明存储数字化波形声音文件名（例如，D：\ MyVc \ danc.wav），这样，waveaudio D：\ MyVc \ danc.wav 才完整地说明了一个可操作的数字化波形设备。

表 7.1 MCI 设备类型及对应的设备驱动程序

| MCI 设备名 | 说明 | 设备驱动程序 |
|---|---|---|
| avivideo | 视频音频交叉设备 | |
| cdaudio | CD 音频设备，如激光唱机 | MCICDA.DRV |
| waveaudio | 波型音频设备，如数字化波形声音设备 | MCIWAVE.DRV |
| sequencer | MIDI 设备，如 MIDI 音序器 | MCISEQ.DRV |
| overlay | 视频叠加设备 | |
| dat | 数字化磁带音频播放机 | |
| digitalvideo | 窗口中的数字视频 | |
| scanner | 图像扫描仪 | |
| vcr | 磁带录像机或播放机 | |
| videodisc | 影碟机 | MCIPIONR.DRV |
| mmmovie | 多媒体影片播放器 | MCIMMP.DRV |
| other | 未定义的 MCI 设备 | |

### 7.2.3 MCI 函数与命令

应用程序通过向 MCI 设备发送命令来控制 MCI 设备。MCI 命令可分为四类，向 MCI 发送命令的方法有两种：

- 调用 MCI 函数 mciSendString()，向 MCI 发送命令字符串。
- 调用 mciSendCommand()函数，向 MCI 发送命令消息。

**1. MCI 命令**

对应于向 MCI 发送命令的两种方法，MCI 命令分为命令消息和命令字符串，主要的 MCI 命令如表 7.2 所示。

表 7.2 MCI 命令列表

| 命令消息 | 命令字符串 | 命令说明 | 命令类型 |
|---|---|---|---|
| MCI_SYSINFO | SYSINFO | 返回有关 MCI 设备的信息 | 系统命令 |
| MCI_BREAK | BREAK | 为一个指定的 MCI 设备设置一个终止键 | |
| MCI_SOUND | SOUND | 播放 Windows 指定的系统声音 | |
| MCI_CLOSE | CLOSE | 关闭一个 MCI 设备 | 通用命令 |
| MCI_GETDEVCAPS | GETDEVCAPS | 获得一个 MCI 设备的性能参数 | |
| MCI_INFO | INFO | 从一个 MCI 设备得到有关的信息 | |
| MCI_OPEN | OPEN | 初始化一个 MCI 设备 | |
| MCI_STATUS | STATUS | 从 MCI 设备返回有关的状态信息 | |

（续表）

| 命令消息 | 命令字符串 | 命令说明 | 命令类型 |
|---|---|---|---|
| MCI _ LOAD | LOAD | 从一个磁盘文件中加载数据 | 可选命令 |
| MCI _ PAUSE | PAUSE | 暂停播放 | |
| MCI _ PLAY | PLAY | 开始播放 | |
| MCI _ RECORD | RECORD | 开始记录数据 | |
| MCI _ RESUME | RESUME | 重新开始播放或录音 | |
| MCI _ SAVE | SAVE | 将数据存储到磁盘文件中 | |
| MCI _ SEEK | SEEK | 向前或向后检索 | |
| MCI _ SET | SET | 设置设备信息 | |
| MCI _ STOP | STOP | 停止播放或记录 | |

**2. MCI 函数**

所有的 MCI 函数名都是以 mci 为前缀，函数原型在 MMSYSTEM.H 中声明。主要的 MCI 函数如表 7.3 所示。

表 **7.3**　常用的 **MCI** 函数

| 函数名 | 功　能 | 类　型 |
|---|---|---|
| mciSendCommand | 发送命令消息 | 向 MCI 发送命令消息函数 |
| mciGetDeviceID | 获得 MCI 设备的 ID | |
| mciSetYieldProc | 设定回调函数 | |
| mciGetYieldProc | 获得回调函数 | |
| mciSendString | 发送命令字符串 | 向 MCI 发送命令字符串函数 |
| mciGetErrorString | 获得当前 MCI 错误的字符串描述 | 公用函数 |

## 7.2.4　MCI 命令字符串接口控制方式

MCI 命令字符串接口控制方式使用 ASCII 字符串来发送驱动 MCI 设备的命令，MCI 的系统、通用和可选命令字符串列于表 7.2，每一个命令字符串都对应着相应的命令消息。同样，这些命令也可具有其相应的扩展形式。

采用这种方式的接口函数有 mciSendString()和 mciGetErrorString()。其中 mciSendString()用于向 MCI 设备发送命令字符串，它们的函数原型如下：

```
MCIERROR mciSendString(
        LPCTSTR    lpszCommand,          //MCI 命令字符串
        LPTSTR     lpszReturnString,     //返回字符串
        UINT       cchReturn,            //返回字符串的大小
        HANDLE     hwndCallback          //回调窗口句柄
        );
BOOL     mciGetErrorString(
        DWORD      fdwError,             //错误代码
```

```
    LPSTR       ErrorMessage,           //错误描述
    UINT        MAXERRORLENGTH          //错误描述长度
    );
```

下面以打开 AVI 数字视频文件 dancing.avi、播放、暂停、恢复直到关闭为例，来说明用 MCI 命令字符串接口控制方式的编程方法。

**1.** 向 **MCI** 发送命令字符串

向 MCI 发送命令字符串的功能函数如下：

```
LONG Execute(LPCSTR strCmd, LPSTR lpstrReturn, BOOL bShowError)
{
   char szReturn[MAX_RETURN_LENGTH];
   //发送 MCI 命令
   MCIERROR mciError = mciSendString(strCmd,              //命令字符串
                                     szReturn,            //返回字符串
                                     sizeof(szReturn),    //字符串长度
                                     m_hWndCallback);     //回调窗口句柄
   lstrcpy(lpstrReturn, szReturn);
   if(bShowError && mciError != 0)                        //出错处理
   {  //获得错误描述字符串
      mciGetErrorString(mciError, szReturn, sizeof(szReturn));
      MessageBox(m_hWndCallback, szReturn,"MCI Error",
                 MB_OK|MB_ICONWARNING);
   }
   return mciError;
}
```

**2.** 打开媒体

打开数字化波形声音、MIDI 音序器和 AVI 数字视频时需要用到别名，打开 AVI 数字视频时还需要控制显示的窗口风格。

例如，打开媒体 dancing.avi 的 MCI 命令字符串为：

```
"open dancing.avi type avivideo alias avivideo_ALIAS"
```

所以，一般的字符串接口控制方式的功能函数为：

```
LONG Open(LPCSTR strMediumName,                 //设备名
          LPCSTR strMediumType,                 //设备类型
          LPCSTR strStyle,                      //显示窗口风格
          HWND hWnd)                            //回调窗口句柄
{
   if(IsMediumOpen())                           //如果已打开
   {
   Stop();                                      //停止播放
   Close();                                     //关闭打开的媒体
   }
   if(hWnd != NULL)m_hWndCallback = hWnd;
```

```
    CString strMedium = strMediumType;
    CString strCmd = "open";
    m_strMediumAlias = strMediumType;
    m_strMediumAlias += "_ALIAS";
    strCmd += strMediumName;
    strCmd += "type";
    strCmd += strMediumType;
    strCmd += "alias";
    strCmd += m_strMediumAlias;

    if(! strMedium.CompareNoCase("AVIVideo"))
    {
       strCmd += "style";
       strCmd += strStyle;
       CString strHWnd;
       if(m_hWndCallback != NULL)
       {
          strHWnd.Format("parent %u", m_hWndCallback);
          strCmd += strHWnd;
       }
    }
    LONG lRet = Execute(strCmd);
    if(lRet == 0)m_bMediumOpen = TRUE;
    return lRet;
}
```

**3.** 播放媒体

播放已打开的媒体的 MCI 命令字符串为：

```
"play avivideo_ALIAS"
```

所以，相应的功能函数为：

```
LONG Play(LPCSTR strStyle, LPCSTR strFlag)
{
    CString strCmd = "play";
    strCmd += m_strMediumAlias;
    CString strStyle0 = strStyle;
    if(! strStyle0.CompareNoCase("fullscreen") ||
        ! strStyle0.CompareNoCase("window"))
    {
       CString strPut = "put";
       strPut += m_strMediumAlias;
       strPut += "source destination";
       Execute(strPut);
       strCmd += strStyle;
    }

    CString s = strFlag;
```

```
    if(! s.IsEmpty())strCmd += " " + strFlag;
    return Execute(strCmd);
}
```

**4.** 暂停播放

暂停播放的 MCI 命令字符串为：

```
"pause avivideo_ALIAS"
```

所以，暂停播放的功能函数为：

```
LONG Pause(void)
{
    CString strCmd = "pause";
    strCmd += m_strMediumAlias;
    return Execute(strCmd);
}
```

**5.** 恢复播放

恢复播放的 MCI 命令字符串为：

```
"resume avivideo_ALIAS"
```

所以，恢复播放的功能函数为：

```
LONG CMCIClass::Resume()
{
    CString strCmd = "resume";
    strCmd += m_strMediumAlias;
    return Execute(strCmd);
}
```

**6.** 停止播放

停止播放的 MCI 命令字符串：

```
"stop avivideo_ALIAS"
```

所以，相应的功能函数为：

```
LONG Stop(void)
{
    CString strCmd = "stop";
    strCmd += m_strMediumAlias;
    return Execute(strCmd);
}
```

**7.** 关闭

关闭的 MCI 命令命令字符串：

```
"close avivideo_ALIAS"
```

所以，相应的功能函数为：

```
LONG CMCIClass::Close()
{
   m_bMediumOpen = FALSE;

   CString strCmd = "close";
   strCmd += m_strMediumAlias;
   return Execute(strCmd);
}
```

### 7.2.5　MCI命令消息接口方式

MCI命令消息接口方式，是利用消息和数据结构来给多媒体设备发送命令和接收MCI设备传来的信息。这种方式的接口函数主要有3个，即mciSendCommand()、mciGetDeviceID()和mciGetErrorString()，前两个函数原型如下：

```
MCIERROR mciSendCommand(
         MCIDEVICEID   wDevice,          //设备 ID
         UINT          uMsg,             //命令消息
         DWORD         fdwCommand ,      //命令消息标志
         DWORD         dwParam           //命令消息使用的结构参数地址
         );
MCIDEVICEID mciGetDeviceID(LPCTSTR lpszDevice);  //设备类型
```

下面以打开AVI数字视频文件dancing.avi、播放、暂停直到关闭为例，来说明用MCI命令消息接口方式的编程方法。

**1.** 预备知识

（1）MCI_OPEN_PARMS、MCI_PLAY_PARMS结构的定义

```
typedef  struct {
         DWORD         dwCallback;       //回调窗口句柄
         MCIDEVICEID   wDevice;          //设备打开成功，返回的设备号
         LPCSTR        lpstrDeviceType;  //设备类型
         LPCSTR        lpstrElementName; //复合设备的设备元素，通常为文件名
         LPCSTR        lpstrAlias;       //指定设备别名
      }MCI_OPEN_PARMS;
typedef  struct {
         DWORD         dwCallback;       //为 MCI_NOTIFY 标志指定一个窗口句柄
         DWORD         dwFrom;           //指定播放起始位置
         DWORD         dwTo;             //指定播放终止位置
      }MCI_PLAY_PARMS;
DWORD    ErrorWord;
```

（2）MCI_OPEN命令消息标志

- MCI_OPEN_ALIAS　MCI_OPEN_PARMS结构的lpstrAlias域中指定了设备别名。
- MCI_OPEN_ELEMENT　MCI_OPEN_PARMS结构的lpstrElementName域中指定的设备

元素。

- MCI _ OPEN _ SHAREABLE 按共享设备方式打开设备。

**2.** 主要功能模块

(1) 初始化一个 MCI 设备

```
MCI _ OPEN _ PARMS mciOpen;
mciOpen.lpstrDeviceType = NULL;
mciOpen.lpstrElementName = "dancing.avi";
ErrorWord = mciSendCommand(0,MCI _ OPEN,MCI _ OPEN _ ELEMENT,
                           (DWORD)(LPVOID)&mciOpen);
if(ErrorWord) 提示 error 信息;}
```

(2) 播放

```
MCI _ PLAY _ PARMS mciPlay;
mciPlay.dwCallback = (long)GetSafeHwnd();
mciPlay.dwFrom = 0;
ErrorWord = mciSendCommand(m _ MCIID,MCI _ PLAY,MCI _ FROM|MCI _ NOTIFY,
                           (DWORD)(LPVOID)&mciPlay);
if(ErrorWord) 提示 error 信息;}
```

(3) 暂停播放

```
ErrorWord = mciSendCommand(m _ MCIID,MCI _ PAUSE,0,NULL);
if(ErrorWord)提示 error 信息;
```

(4) 重新开始播放

```
ErrorWord = mciSendCommand(m _ MCIID,MCI _ RESUME,0,NULL);
```

(5) 停止播放

```
ErrorWord = mciSendCommand(m _ MCIID,MCI _ STOP,MCI _ WAIT,NULL);
```

(6) 关闭一个 MCI 设备

```
ErrorWord = mciSendCommand(m _ MCIID,MCI _ CLOSE,MCI _ WAIT,NULL);
if(ErrorWord)提示 error 信息;
```

## 7.3 构建 CMCIClass 类

上节介绍的 MCI 的基本编程方法，向 MCI 发送命令的两种方法，都必须很烦琐地设置很多参数，使用起来很不方便。本节将定义一个类 CMCIClass，把所有 MCI 命令字符串接口控制方式的 MCI 功能进行封装，隐藏所有的 MCI 命令参数，仅对外公布简单的操作接口。使用 CMCIClass 类简化 MCI 编程方式。

### 7.3.1　CMCIClass 类的成员构成

（1）父类

由于不需要其他类的特征，因此将 CMCIClass 设计为无父类的基类。

（2）数据成员

用于描述类的内部性质，为了在其派生类中可见，设计为 protected 类型：

① 设备别名：　CString　m _ strMediumAlias。

② 回调窗口句柄：　HWND　m _ hWndCallback。

③ 媒体打开标记：　BOOL　m bMediumOpen。

（3）函数成员

主要提供 MCI 操作接口：

① 打开媒体文件。

② 播放。

③ 暂停。

④ 关闭。

⑤ 其他。

要求 MCI 操作函数都返回一个长整型，用于表示命令执行的状态，如为非零，则可用 mciGetErrorString 获取错误描述。

### 7.3.2　CMCIClass 类的定义

CMCIClass 的定义如下：

```
//MCIClass.h
class CMCIClass
{protected:
      //数据成员，仅用于描述类的内部性质，为了在其派生类中可见
      CString m _ strMediumAlias;          //设备别名
      HWND m _ hWndCallback;               //回调窗口句柄
      BOOL m _ bMediumOpen;                //媒体是否打开标记
public:
      CMCIClass(HWND hWnd = NULL);
      virtual ~CMCIClass();

      LONG Open(LPCSTR strMedium, LPCSTR strMediumType, LPCSTR strStyle = " ", HWND
hWnd = NULL);
      LONG Close();
      LONG Play(LPCSTR strStyle = " ", LPCSTR strFlag = " ");
LONG  Pause();
      LONG Resume();
      LONG Stop();
LONG  Seek(LONG lPos);
      LONG Seek(LPCSTR strPos);
```

```
        LONG SetTimeFormat(LPCSTR strTimeFormat = "ms");
        LONG GetLength();

        CString GetTimeFormat();
        CString GetMode();
        LONG GetPosition(LPSTR strPos);
        LONG GetStartPosition(LPSTR strPos);
        BOOL IsMediumOpen();
        BOOL   IsMediumPresent();

        LONG Execute(LPCSTR strCmd, LPSTR lpstrReturn = NULL, BOOL bShowError =
FALSE);
        CString GetMediumAlias();
        HWND GetCallbackHWnd();
        LONG SetAudioVolume(int nVolumeFactor);
};
```

### 7.3.3 CMCIClass 类的实现

```
// MCIClass.cpp: implementation of the CMCIClass class.
//
//////////////////////////////////////////////////////////////////////

#include "stdafx.h"
#include <math.h>
#include <mmsystem.h>

//#include "MCIPlayer.h"
#include "MCIClass.h"

#ifdef _DEBUG
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#define new DEBUG_NEW
#endif

//////////////////////////////////////////////////////////////////////
// Construction/Destruction
//////////////////////////////////////////////////////////////////////

CMCIClass::CMCIClass(HWND hWnd)
{
   m_strMediumAlias = " ";
   m_hWndCallback = hWnd;
   m_bMediumOpen = FALSE;
}
```

```
CMCIClass::~CMCIClass()
{
   Stop();
   Close();
}

LONG CMCIClass::Open(LPCSTR strMediumName,
                     LPCSTR strMediumType,
                     LPCSTR strStyle,
                     HWND hWnd)
{
   if(IsMediumOpen())
   {
      Stop();
      Close();
   }

   if(hWnd != NULL)
       m_hWndCallback = hWnd;

   CString strMedium = strMediumType;
   CString strCmd = "open";
    m_strMediumAlias = strMediumType;
      m_strMediumAlias += "_ALIAS";

       strCmd += strMediumName;
       strCmd += "type";
       strCmd += strMediumType;
       strCmd += "alias";
       strCmd += m_strMediumAlias;

   if(! strMedium.CompareNoCase("AVIVideo"))
   {
      strCmd += "style";
      strCmd += strStyle;

      CString strHWnd;
      if(m_hWndCallback != NULL)
      {
         strHWnd.Format("parent %u", m_hWndCallback);
         strCmd += strHWnd;
      }
   }

   LONG lRet = Execute(strCmd);

   if(lRet == 0)
       m_bMediumOpen = TRUE;
```

```
    return lRet;
}

LONG CMCIClass::Close()
{
    m_bMediumOpen = FALSE;

    CString strCmd = "close";
    strCmd += m_strMediumAlias;

    return Execute(strCmd);
}

LONG CMCIClass::Play(LPCSTR strStyle, LPCSTR strFlag)
{
    CString strCmd = "play";
    strCmd += m_strMediumAlias;

    CString strStyle0 = strStyle;
    if(! strStyle0.CompareNoCase("fullscreen") ||
        ! strStyle0.CompareNoCase("window"))
    {
        CString strPut = "put";
        strPut += m_strMediumAlias;
        strPut += "source destination";
        Execute(strPut);
        strCmd += strStyle;
    }

    CString s = strFlag;
    if(! s.IsEmpty())
    {
        strCmd += " ";
        strCmd += strFlag;
    }

    return Execute(strCmd);
}

LONG CMCIClass::Pause()
{
    CString strCmd = "pause";
    strCmd += m_strMediumAlias;

    return Execute(strCmd);
}
```

```
LONG CMCIClass::Resume()
{
   CString strCmd = "resume";
   strCmd += m_strMediumAlias;

   return Execute(strCmd);
}

LONG CMCIClass::Record()
{
   CString strCmd = "record";
   strCmd += m_strMediumAlias;

   return Execute(strCmd);
}

LONG CMCIClass::Save(LPCSTR strFileName)
{
   CString strCmd = "save";
   strCmd += m_strMediumAlias + " ";
   strCmd += strFileName;

   return Execute(strCmd);
}

LONG CMCIClass::Seek(LONG lPos)
{
   CString strPos;
   if(lPos == 0L)
        strPos = "to start";
   else if(lPos == -1L)
        strPos = "to end";
   else
        strPos.Format("to %ld", lPos);

   CString strCmd = "seek";
   strCmd += m_strMediumAlias;
   strCmd += strPos;

   return Execute(strCmd);
}

LONG CMCIClass::Seek(LPCSTR strPos)
{
   CString strCmd = "seek";
   strCmd += m_strMediumAlias;
   strCmd += "to";
   strCmd += strPos;
```

```
    return Execute(strCmd);
}

LONG CMCIClass::Stop()
{
    CString strCmd = "stop";
    strCmd += m_strMediumAlias;

    return Execute(strCmd);
}

LONG CMCIClass::SetTimeFormat(LPCSTR strTimeFormat)
{
    CString strCmd = "set";
    strCmd += m_strMediumAlias;
    strCmd += "time format";
    strCmd += strTimeFormat;

    return Execute(strCmd);
}

// get the medium length in ms unit
LONG CMCIClass::GetLength()
{
    CString strCmd = "status";
    strCmd += m_strMediumAlias;
    strCmd += "length";

    // kepp old time format
    CString strTimeFormat = GetTimeFormat();
    // set time format to ms
    SetTimeFormat("ms");

    char szReturn[MAX_RETURN_LENGTH];
    Execute(strCmd, szReturn);

    // restore old time format
    SetTimeFormat(strTimeFormat);

    return atol(szReturn);
}

CString CMCIClass::GetTimeFormat()
{
    CString strCmd = "status";
    strCmd += m_strMediumAlias;
    strCmd += "time format";
```

```
    char szReturn[MAX_RETURN_LENGTH];
    Execute(strCmd, szReturn);

    return CString(szReturn);
}

CString CMCIClass::GetMode()
{
    if (!IsMediumOpen())
        return CString("not open");

    CString strCmd = "status";
    strCmd += m_strMediumAlias;
    strCmd += "mode";

    char szReturn[MAX_RETURN_LENGTH];
    Execute(strCmd, szReturn);

    return CString(szReturn);
}

LONG CMCIClass::GetPosition(LPSTR strPos)
{
    CString strCmd = "status";
    strCmd += m_strMediumAlias;
    strCmd += "position";

    char szReturn[MAX_RETURN_LENGTH];
    Execute(strCmd, szReturn);

    lstrcpy(strPos, szReturn);

    return atol(szReturn);
}

LONG CMCIClass::GetStartPosition(LPSTR strPos)
{
    CString strCmd = "status";
    strCmd += m_strMediumAlias;
    strCmd += "start position";

    char szReturn[MAX_RETURN_LENGTH];
    Execute(strCmd, szReturn);

    lstrcpy(strPos, szReturn);

    return atol(szReturn);
```

```
}
BOOL CMCIClass::IsMediumOpen()
{
   return m_bMediumOpen;
}

BOOL CMCIClass::IsMediumPresent()
{
   CString strCmd = "status";
   strCmd += m_strMediumAlias;
   strCmd += "media present";

   char szReturn[MAX_RETURN_LENGTH];
   Execute(strCmd, szReturn);

   if (stricmp(szReturn,"true"))
        return FALSE;
   else
        return TRUE;
}

LONG CMCIClass::Execute(LPCSTR strCmd, LPSTR lpstrReturn, BOOL bShowError)
{
   char szReturn[MAX_RETURN_LENGTH];
   MCIERROR mciError = mciSendString(strCmd,
                                     szReturn,
                                     sizeof(szReturn),
                                     m_hWndCallback);
   lstrcpy(lpstrReturn, szReturn);
   if (bShowError && mciError != 0)
   {
      mciGetErrorString(mciError, szReturn, sizeof(szReturn));
      MessageBox(m_hWndCallback, szReturn,"MCI Error", MB_OK|MB_ICONWARNING);
   }

   return mciError;
}

CString CMCIClass::GetMediumAlias()
{
   return m_strMediumAlias;
}

HWND CMCIClass::GetCallbackHWnd()
{
   return m_hWndCallback;
}
```

# 7.4　媒体播放器

有了前面的准备工作，接下来将利用 MCIClass 类编写一个基于对话框的播放器，尝试开发媒体播放器的乐趣。

## 7.4.1　创建工程

使用应用向导创建一个基于对话框的应用程序，工程名为 MyPlayer。创建步骤如下：

（1）启动 Visual C++6.0。从“File”菜单中选择“New”。此时，Visual C++将显示一个“New”对话框。

（2）在“New”对话框中选择“Project”标签，然后指定工程类型 MFC AppWizard[exe]、工程名 MyPlayer 和工程位置 d:\MYVC。

（3）单击“OK”按钮，此时 Visual C++将显示“MFC AppWizard - Step 1”对话框。

（4）在“MFC AppWizard - Step 1”对话框中，选择“Dialog based”，创建一个基于对话框的应用程序。

（5）单击“Finish”按钮，此时 Visual C++将显示“NewProject Information”窗口。

（6）单击“OK”，于是，Visual C++就会创建 MyPlayer 工程以及相关的所有文件。

## 7.4.2　可视化设计

编辑对话框资源的步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 MyPlayer resources 扩展开，然后再扩展 Dialog，最后，双击“IDD_MYPLAYER_DIALOG”项。

（2）从“IDD_MYPLAYER_DIALOG”对话框中删除“OK”、“Cancel”按钮以及 TODO 文本。根据表 7.4 中的定义，编辑对话框资源“IDD_MYPLAYER_DIALOG”。编辑后的对话框如图 7.2 所示。

表 7.4　IDD_MYPLAYER_DIALOG 对话框的属性表

| 对　象 | 属　性 | 属 性 值 |
|---|---|---|
| Group Box | ID，Caption | IDC_STATIC，选择媒体 |
| Radio Button | ID，Caption，Group | IDC_RADIO_WAVE，WAVE，Checked |
| Radio Button | ID，Caption | IDC_RADIO_MIDI，MIDI |
| Radio Button | ID，Caption | IDC_RADIO_AVI，AVI |
| Button | ID，Caption | IDC_BUTTON_OPEN，打　开 |
| Button_ | ID，Caption | IDC_BUTTON_PLAY，播　放 |
| Button | ID，Caption | IDC_BUTTON_STOP，停　止 |

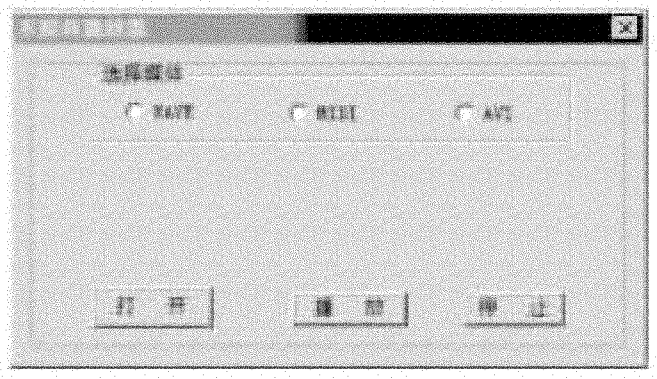



图 7.2 编辑完后的“IDD _ MYPLAYER _ DIALOG”对话框资源

### 7.4.3 将 CMCIClass 类插入工程

为了在 CMyPlayerDlg 类中直接使用 CMCIClass 类的成员函数，必须将 CMCIClass 类插入工程 MyPlayer 并作为 CMyPlayerDlg 类的父类。

**1.** 将 **CMCIClass** 类插入工程

具体操作步骤的是：

（1）将 CMCIClass 类的构成文件 MCIClass.h 和 MCIClass.cpp 拷贝到本工程所在目录中。

（2）选择“Project”菜单下的“Add to Project”菜单项。

（3）在级联菜单中，选择“Files”项，弹出“Insert Files into Project”对话框，搜索到 MyPlayer 工程，如图 7.3 所示。

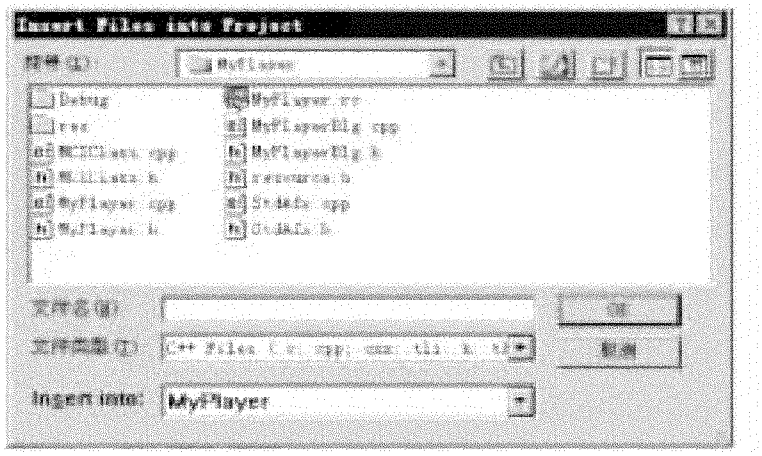



图 7.3 “Insert Files into Project”对话框

（4）同时选中 MCIClass.h 和 MCIClass.cpp 文件，单击“OK”按钮，就将 MCIClass.h 和 MCIClass.cpp 文件插入了 MyPlayer 工程。

**2.** 修改 **CMyPlayerDlg** 的基类

在 MyPlayerDlg.h 文件中添加包含 MCIClass.h 头文件语句，修改 CMyPlayerDlg 类的的定义，使其成为 CDialog 和 CMCIClass 的共同子类。

```
// MyPlayerDlg.h : header file
......
#include"MCIClass.h"
class CMyPlayerDlg : public CDialog,protected CMCIClass
{
// Construction
public:
    CMyPlayerDlg(CWnd * pParent = NULL); // standard constructor
    ......
    DECLARE_MESSAGE_MAP()
};
```

### 7.4.4　为“WAVE” Radio 控件引入变量

媒体播放器主窗口对话框中的3个“Radio”单选按钮，用于标识媒体类型：波形音频文件、音频CD和视频，所以，必须关联一个变量。操作方法如下：

（1）在“View”中选择“ClassWizard”，然后在“Member Variable”标签，进行如下设置：

```
Class name: CMyPlayerDlg
Control Ids: IDC_RADIO_WAVE
```

（2）单击“Add Variable...”按钮，此时，Visual C++将显示一个“Add Member Variable”对话框。

（3）设置下面列出的选项：

```
Member Variable name:   m_radio
Category:   value
Variable type:   int
```

（4）单击“Add Member Variable”对话框的“OK”按钮。返回到“MFC ClassWizard”对话框，单击“MFC ClassWizard”对话框的“OK”按钮，于是，Visual C++就为Radio控件“IDC_RADIO_WAVE”添加了一个成员变量m_radio，如图7.4所示。

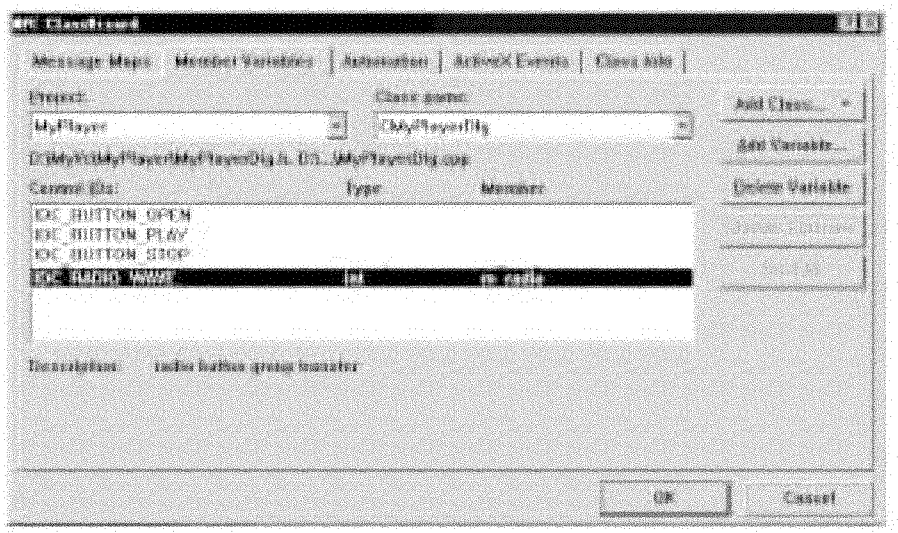


图7.4　为控件“IDC_RADIO_WAVE”引入变量m_radio

### 7.4.5 为 Button 按钮的 BN _ CLICKED 事件编写代码

**1.** 为 **Button** 按钮的 **BN _ CLICKED** 事件添加消息响应函数

用“MFC ClassWizard”，分别为“打开”、“播放”和“停止”3 个 Button 按钮的 BN _ CLICKED 事件添加消息响应函数 OnRadioWave()、OnRadioMidi()和 OnRadioAvi()，如图 7.5 所示。

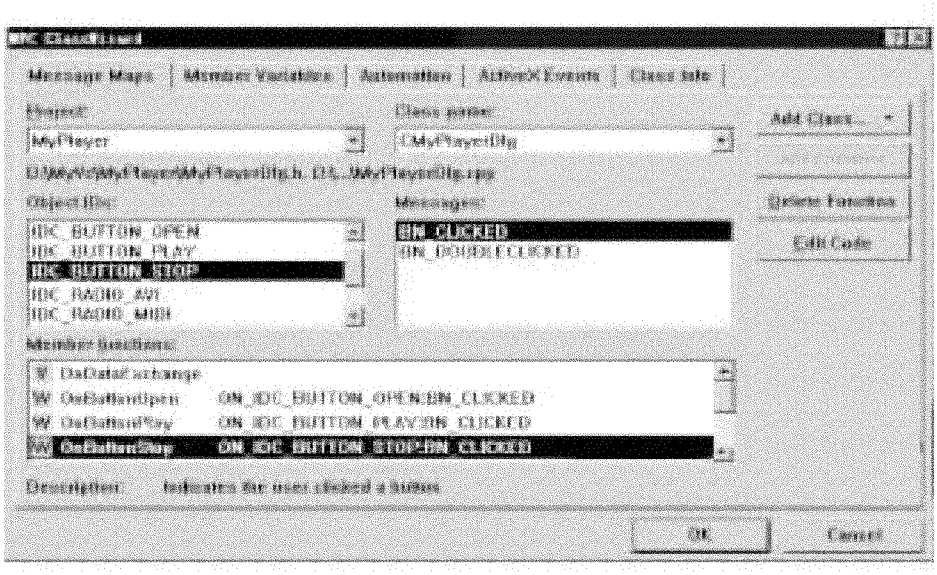



图 7.5 Button 按钮添加消息响应函数

**2.** 编写程序代码

（1）为“打开”按钮的消息响应函数 OnButtonOpen()编写如下代码：

```
void CMyPlayerDlg::OnButtonOpen()
    {
      CString strFilter;
    UpdateData(true); //更新 Radio 单选按钮关联变量 m_radio
      switch (m_radio)
    {
      case 0: strFilter = "Wave Files (*.wav)|*.wav|All Files (*.*)|*.*||";
              break;
      case 1: strFilter = "Midi Files (*.mid)|*.mid|All Files (*.*)|*.*||";
              break;
      case 2: strFilter = "AVI Files (*.avi)|*.avi|All Files (*.*)|*.*||";
              break;
    }
    CFileDialog FileDlg(true, NULL, NULL,
                        OFN_HIDEREADONLY|OFN_OVERWRITEPROMPT,
                        (LPCSTR)strFilter, this);
    if (FileDlg.DoModal() == IDOK)
    {
      CString strFileName = FileDlg.GetPathName();
```

```
        char szFileName[_MAX_PATH];
        GetShortPathName((LPCSTR)strFileName, szFileName, _MAX_PATH);
        char szExt[8];
        _splitpath((const char *)szFileName, NULL, NULL, NULL, szExt);
        // keep the open result
        MCIERROR mciError = 0;
        if(! stricmp(szExt,".wav"))
            mciError = Open(szFileName,"waveaudio"," ", m_hWnd);
        else if(! stricmp(szExt,".mid"))
            mciError = Open(szFileName,"sequencer"," ", m_hWnd);

        else if(! stricmp(szExt,".avi"))
            mciError = Open(szFileName,"avivideo","overlapped", m_hWnd);
        // our default time format is ms
        if(mciError == 0)
        {
           SetTimeFormat("ms");
        }

        // set new title
        CString s;
        s.LoadString(IDS_TITLE);
        s += " - ";
        s += FileDlg.GetFileName();
        SetWindowText(s);
        }
}
```

（2）为“播放”按钮的消息响应函数 OnButtonPlay()编写如下代码：

```
void CMyPlayerDlg::OnButtonPlay()
{   Play();
}
```

（3）为“停止”按钮的消息响应函数 OnButtonStop()编写如下代码：

```
void CMyPlayerDlg::OnButtonStop()
{
   Stop();
}
```

### 7.4.6　按钮状态更新

类似于菜单项的状态更新，有时，需要禁止某个 Button 按钮，使之成为非活动状态，不允许用户点击。

Button 按钮也是一个窗口，要使之成为非活动状态，可用窗口类的成员函数 EnableWindow()来控制窗口的状态。通过指向该窗口的指针来控制 Button 按钮的活动状态。而 GetDlgItem(UINT pID)函数就是用于获取参数中指定窗口的指针的。

**1.** 设置 **Button** 按钮的初始状态

在 OnInitDialog()函数中，设置“打开”、“播放”和“停止”按钮的初始状态。

```
BOOL CMyPlayerDlg::OnInitDialog()
{
   CDialog::OnInitDialog();
   ......
   SetIcon(m_hIcon, TRUE);          // Set big icon
   SetIcon(m_hIcon, FALSE);         // Set small icon
   //设置 Button 按钮的初始状态
   GetDlgItem(IDC_BUTTON_OPEN)->EnableWindow(true);
   GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(FALSE);
   GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);

   return TRUE; // return TRUE unless you set the focus to a control
}
```

**2.** 在点击“播放”后，更新“播放”和“停止”按钮状态

修改“播放”按钮的消息响应函数，使之在用户点击“播放”按钮后，更新“播放”和“停止”按钮状态。

```
void CMyPlayerDlg::OnButtonPlay()
{
   GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(true);
   GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(false);
   Play();
}
```

### 7.4.7 修改工程设置、构建并运行程序

**1.** 修改工程设置

从“Project”菜单中选择“Setting”菜单项，弹出“Project Settings”对话框。选中“Link”标签，在“Object/library modules”编辑框中输入 winmm.lib，如图 7.6 所示。单击“OK”按钮，就为工程 MyPlayer 添加了多媒体库 winmm.lib 的连接设置。

**2.** 查看结果

(1) 编译、连接和运行程序，出现如图 7.2 所示的媒体播放器前面。

(2) 点击 AVI 单选按钮，设置 AVI 类型媒体。

(3) 单击“打开”按钮，在弹出的通用文件对话框中，选择待播放的媒体文件(例如，Dancing.avi)，单击“OK”按钮。

(4) 单击“播放”按钮，开始播放所选媒体文件，如图 7.7 所示。

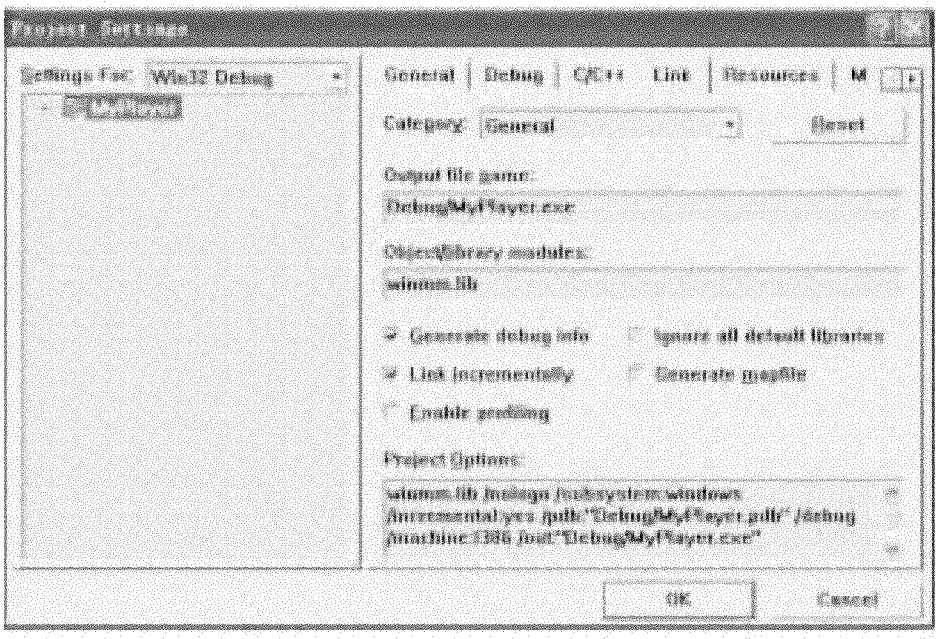



图 7.6　“Project Settings”对话框


图 7.7　播放器的运行界面

### 7.4.8　技术要点

**1. Radio Button 控件**

一般将几个 Radio Button(单选按钮)控件组合起来使用,表示从几个选项中挑选出一个。

例如,如果要将 3 个 Radio Button 控件组合成一组,首先要将这 3 个 Radio Button 控件的挑格顺序连在一起,然后选中第一个控件的 General 标签的 Group 属性,确认第二、三个 Radio Button 控件的 Group 属性没有选中。最后还要确认紧接着的一个控件(如果有的话)的 Group 属性被选中。

把 Radio Button 控件组合好之后,打开“MFC ClassWizard”对话框。在“Member Variables”标签中选中 MyPlayerDlg 类后,会发现只有第一个(Group 属性被选中的)Radio Button 控件的标识符 IDC_RADIO_WAVE 出现在“Control IDs”项中。

利用 MFC ClassWizard 为 IDC_RADIO_WAVE 引入 int 型关联变量 m_radio,该变量的数值从 0 逐渐增大表示相应的 Radio Button 控件被选中。即:

- 当该变量为 0 时，表示第 1 个 Radio Button 控件(WAVE)被选中。
- 当该变量为 1 时，表示第 2 个 Radio Button 控件(MIDI 被选中。
- 当该变量为 2 时，表示第 3 个 Radio Button 控件(AVI)被选中。

**2. EnableWindow(BOOL nID)** 函数

窗口类的成员函数 EnableWindow()，其功能是用于控制窗口的状态。

其中参数 nID：

为 TRUE 时，窗口处于活动状态；

为 FALSE 时，窗口处于非活动状态，即被禁止。

例如，GetDlgItem(IDC _ BUTTON _ OPEN) -> EnableWindow(TRUE)；使“打开”按钮处于活动状态，其中 IDC _ BUTTON _ OPEN 是“打开”按钮的标识号。

**3. ShowWindow (BOOL nID)** 函数

窗口类的成员函数 ShowWindow ()，其功能是用于控制窗口的可见性。

其中参数 nID：

为 SW _ SHOW 时，窗口处于可见；

为 SW _ HIDE 时，窗口处于隐藏。

例如，GetDlgItem(IDC _ BUTTON _ OPEN) -> ShowWindow(SW _ SHOW)使“打开”按钮处于可见；而 GetDlgItem(IDC _ BUTTON _ PLAY) -> ShowWindow(SW _ HIDE)隐藏“打开”按钮。

其中 IDC _ BUTTON _ OPEN 是“打开”按钮的标识号。

**4. GetDlgItem()** 函数

GetDlgItem() 函数的原型为：

```
Wnd * GetDlgItem(UINT pID);
```

其中参数 pID 为控件的标识号(ID)，函数将返回指向标识号为 pID 控件的指针。常用于控制控件的状态和可见性。例如：

要设置控件状态，代码为：

```
GetDlgItem(IDC _ BUTTON _ OPEN) -> EnableWindow(true);     //激活
GetDlgItem(IDC _ BUTTON _ PLAY) -> EnableWindow(FALSE);    //禁止
```

要设置控件的可见性，代码为：

```
GetDlgItem(IDC _ BUTTON _ OPEN) -> ShowWindow(SW _ SHOW); //显示
GetDlgItem(IDC _ BUTTON _ PLAY) -> ShowWindow(SW _ HIDE); //隐藏
```

## 7.5 为 MyPlayer 添加进程条

进程条是一种经常使用的控件，称为 Progress 控件，可用来显示事务的进程。它是一个用色块缓慢填充的矩形框，色块填充越多，任务越接近尾声。当进程条完全填满时，与之相关的任务

也就完成了。

一般当一项进程操作的时间销长(几秒钟以上),而又没有其他信息向用户显示的时候,最好制作一个进程条,以显示事务的进程。下面将为多媒体播放器 MyPlayer 添加进程条,以显示播放文件的进程。

### 7.5.1　进程条的可视化设计

为 MyPlayer 程序添加进程条的操作步骤如下:

(1) 在"Project Workspace"窗口中,单击"ResourceView"标签,把 MyPlayer resources 扩展开,然后再扩展 Dialog,然后双击"IDD _ MPLAYER _ DIALOG"项。在右边的工作台中显示可以进行可视化编辑的对话框,并打开一个控件工具窗口。

(2) 在"IDD _ MPLAYER _ DIALOG"对话框中,添加表 7.5 所示的"Progress"和"Static Text"控件。设计完毕对话框如图 7.8 所示。

表 7.5　"IDD _ MPLAYER _ DIALOG"对话框新添控件的属性表

| 对象 | 属性 | 属性值 |
|---|---|---|
| Progress Contron | ID<br>Client edge, Static edge<br>Border, Smooth | IDC _ PROGRESS<br>Checked(Extended Styles tab)<br>Checked( Styles tab) |
| Static Text | ID | IDC _ EDIT _ DIGHT |

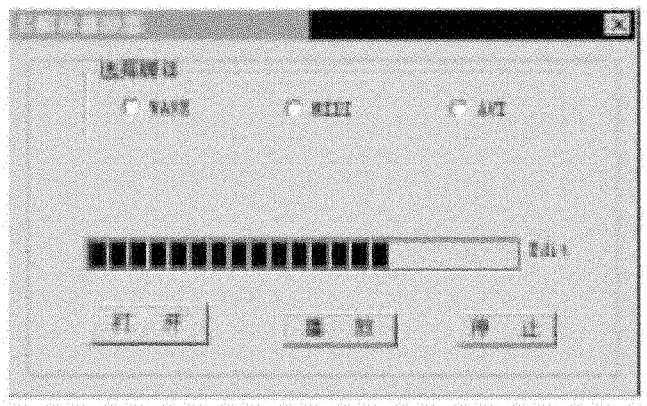


图 7.8　添加了进程条控件的 MyPlayer 播放器对话框

### 7.5.2　为 Progress 控件引入变量

为了在播放过程中,用进程条来显示媒体播放的进程,必须为之关联一个变量。给"IDC _ PROGRESS"控件引入变量的步骤如下:

(1) 在"View"中选择"ClassWizard",然后在"Member Variable"标签中,进行如下设置:

```
Class name: CMyPlayerDlg
Control IDs: IDC _ PROGRESS
```

（2）单击“Add Variable...”按钮，此时，Visual C++将显示一个“Add Member Variable”对话框。

（3）设置下面列出的选项：

```
Variable name: m_Progress
Category: control
Variable type: CProgressCtrl
```

（4）单击“Add Member Variable”对话框的“OK”按钮。

（5）单击“MFC ClassWizard”对话框的“OK”按钮，Visual C++就为进程条控件“IDC_PROGRESS”添加了一个成员变量 m_Progress，如图 7.9 所示。

### 7.5.3 为 Static Text 控件引入变量

为了在播放过程中，用“IDC_EDIT_DIGHT”控件显示进程的百分数，必须为该控件引入一个变量。给“IDC_EDIT_DIGHT”控件引入变量的步骤如下：

（1）在“View”中选择“ClassWizard”，然后在“Member Variable”标签中，进行如下设置：

```
Class name: CMyPlayerDlg
Control IDs: IDC_EDIT_DIGHT
```

（2）单击“Add Variable...”按钮，此时，Visual C++将显示一个“Add Member Variable”对话框。

（3）设置下面列出的选项：

```
Variable name: m_dight
Category: value
Variable type: CString
```

（4）单击“Add Member Variable”对话框的“OK”按钮。

（5）单击“MFC ClassWizard”对话框的“OK”按钮，Visual C++就为 Static Text 控件“IDC_EDIT_DIGHT”添加了一个成员变量 m_dight，如图 7.9 所示。

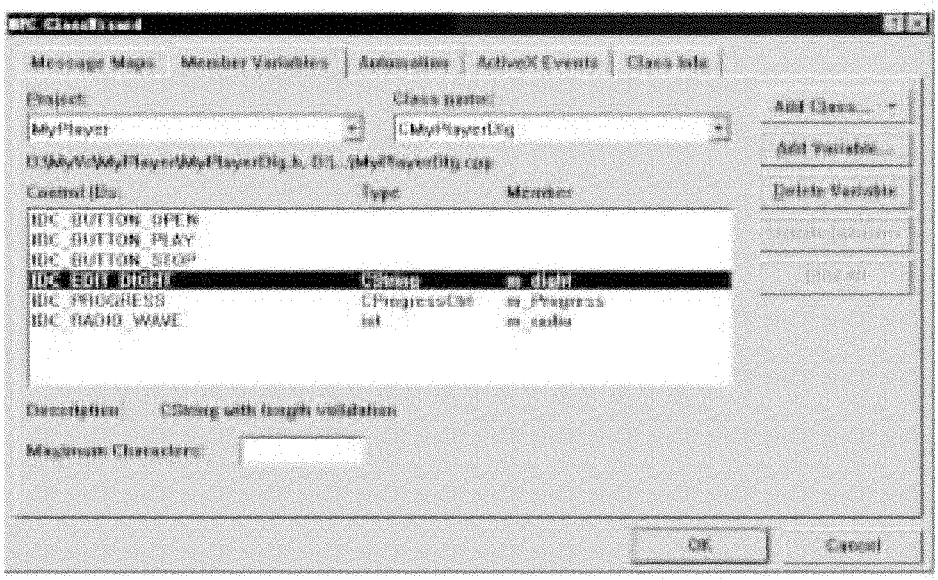



图 7.9 为控件“IDC_EDIT_DIGHT”引入变量

### 7.5.4　初始化进程条、设置定时器

定位到函数 CMyPlayerDlg::OnInitDialog()，设置定时器，初始化进程条的范围。

```
BOOL CMplayerDlg::OnInitDialog()
{
   Cdialog::OnInitDialog();
   ...
   m_Progress.SetRange(0,100);              // 初始化进程条的范围
   m_Progress.ShowWindow(SW_HIDE);          // 隐藏进程条
   SetTimer(100,500,NULL);                  //设置定时器
   return TRUE;      // return TRUE unless you set the focus to a control
}
```

### 7.5.5　操作进程条

为了利用进程条来显示播放文件的进程，需添加定时器的响应函数 OnTimer()，其构造方法如下：

（1）在“View”中选择“ClassWizard”，然后在“Member Maps”标签中进行如下设置：

```
Class name: CMyPlayerDlg
Object IDs: CMyPlayerDlg
Message: WM_TIMER
```

（2）单击“Add Function”按钮，创建 OnTimer()新函数。

（3）单击“Edit Code”按钮，在 OnTimer()函数中编辑如下代码：

```
void CMyPlayerDlg::OnTimer(UINT nIDEvent)
{
   CString strMode = GetMode();
   if(strMode.CompareNoCase("not open"))
   {
      if(IsMediumPresent())
      {
         if(! strMode.CompareNoCase("stopped"))
            {
               GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(TRUE);
               GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
            }
         else if(! strMode.CompareNoCase("not ready"))
         {
            GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(FALSE);
            GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
         }
         char szPos[256];
         m_Progress.SetPos((int)(100 * GetPosition(szPos)/GetLength()));
         m_dight = itoa((100 * GetPosition(szPos)/GetLength()),szPos,10);
```

```
            m_dight += "%";
            UpdateData(false);
        }
    else
        {
            GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(FALSE);
            GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
        }
    }
    CDialog::OnTimer(nIDEvent);
}
```

### 7.5.6 修改进程条可见属性

修改 OnPlay( ) 函数，使其开始播放媒体时，设置进程条可见。

```
void CMplayerDlg::OnPlay()
{
    m_Progress.ShowWindow(SW_SHOW);     //设置进程条可见
    Play();
}
```

### 7.5.7 构造并运行 MyPlayer

编译、连接和执行 MyPlayer 程序，点击“AVI”单选按钮，设置 AVI 类型媒体，再单击“打开”按钮，在“打开”对话框中，选取要播放的 *.avi 文件，最后单击“播放”按钮，开始播放所选的媒体文件，并显示播放进度，如图 7.10 所示。


图 7.10 含有进程条的播放器（正在播放 Dancing.avi 文件）

### 7.5.8 技术要点

**1. Progress 控件**

进程条称为 Progress 控件，可用来显示事务的进程。一般当一项进程操作的时间稍长（几秒

钟以上),而又没有其他信息向用户显示的时候,最好制作一个进程条,以显示事务的进程。

Progress 控件与 CProgressCtrl 类相关联。CProgressCtrl 类的常用成员函数如表 7.6 所示。

表 **7.6**　**CProgressCtrl** 类的常用成员函数

| 函数原型 | 说　明 |
|---|---|
| void SetRange(short nLower, short nUpper) | 设置进程条的数值范围 |
| int SetPos(int nPos) | 设置进程条上当前的位置 |
| int SetStep(int nStep) | 设置进程条前进步长 |
| int OffsetPos(int nPos) | 在进程条上偏移一定的位置 |

**2.** 进程条的初始化

为了实现进程条显示播放文件的进程,需要对进程条初始化、设置定时器和同步操纵进程条。函数 OnInitDialog()中的下列部分代码:

```
SetTimer(100,500,NULL);
m_Progress.SetRange(0,100);
m_Progress.SetPos(0);
m_Progress.ShowWindow(SW_HIDE);  // 隐藏进程条
```

就是对进程条进行初始化。其中:

- 语句 SetTimer(100,500,NULL)用以设置定时器,以定时更新进程条的进程显示。
- 语句 m_Progress.SetRange(0,100) 设置进程条的数值范围,适当而不惟一。
- m_Progress.SetPos(0)设置进程条初始位置。
- m_Progress.ShowWindow(SW_HIDE)隐藏进程条。

**3.** 操作进程条

在定时器的响应函数 OnTimer()中,语句:

```
char szPos[256];
m_Progress.SetPos((int)(100 * GetPosition(szPos)/GetLength()));
```

用以更新进程条的显示,其中 GetLength()是获取播放文件的长度,因此,数值(100 * GetPosition(szPos)/GetLength())为相对进程条的最大值的相对数值。

## 7.6　为 MyPlayer 添加滑动条

类似于进程条,滑动条也是一种经常使用的控件,称为 Slider 控件,可用来显示事务的进程。它由刻度和“滑块”共同构成,可由用户通过鼠标或箭头键来控制,以图形方式从一定的取值范围中选取一个数值。

### 7.6.1 滑动条的可视化设计

对话框为 MyPlayer 程序添加滑动条控件的操作步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 MyPlayer resources 扩展开，然后再扩展 Dialog，然后双击“IDD _ MYPLAYER _ DIALOG”项。在右边的工作台中显示可以进行可视化编辑的对话框，并打开一个控件工具窗口。

（2）在“IDD _ MYPLAYER _ DIALOG”对话框中添加 Slider 控件。

（3）根据表 7.7，设计 Slider 控件“IDC _ SLIDER”。设计完毕的媒体播放器对话框如图 7.11 所示。

表 7.7 “IDD _ MYPLAYER _ DIALOG”对话框的属性表

| 对象 | 属性 | 属性值 |
|---|---|---|
| Slider Contron | ID<br>Client edge<br>Tick marks，Auto ticks，Border<br>Orientation<br>Point | IDC _ SLIDER<br>Checked（Extended Styles tab）<br>Checked（ Styles tab）<br>Horizonta<br>Top/Left |


图 7.11 添加了滑动条控件的 MyPlayer 播放器对话框

### 7.6.2 为 Slider 控件引入变量

为了在播放过程中，用滑动条控件来显示媒体播放的进程，必须为之关联一个变量。给“IDC _ SLIDER”控件引入变量的步骤如下：

（1）在“View”中选择“ClassWizard”标签，然后在“Member Variable”标签中，进行如下设置：

```
Class name: CMyplayerDlg
Control Ids: IDC _ SLIDER
```

（2）单击“Add Variable...”按钮，此时，Visual C++ 将显示一个“Add Member Variable”对话框。

（3）设置下面列出的选项：

```
Variable name: m_Slider
Category: control
Variable type: CSliderCtrl
```

（4）单击“Add Member Variable”对话框的“OK”按钮。于是，Visual C++就为滑动条控件“IDC_SLIDER”添加了一个成员变量 m_Slider，如图 7.12 所示。

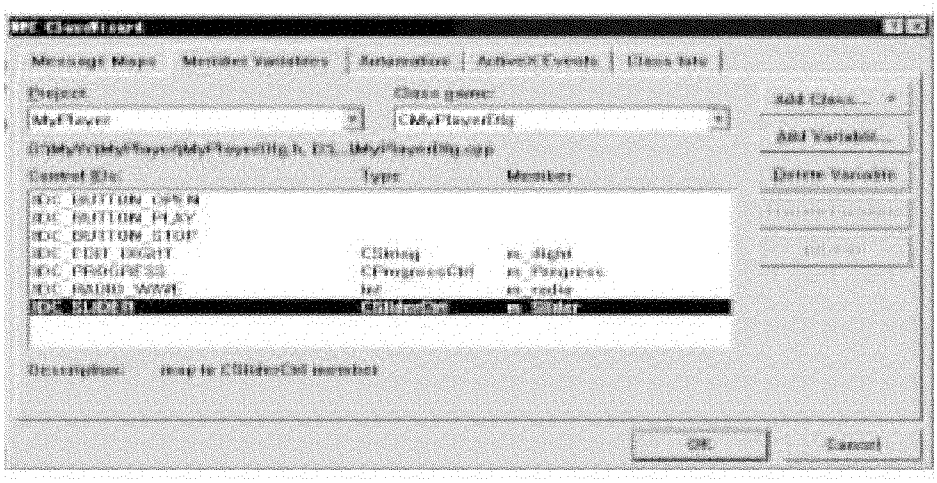


图 7.12　为滑动条控件“IDC_SLIDER”添加了一个成员变量 m_Slider

### 7.6.3　初始化 Slider 控件

在对话框的实现文件中，添加滑动条控件的初始化代码到 OnInitDialog()中。

```
BOOL CMyPlayerDlg::OnInitDialog()
{
     CDialog::OnInitDialog();
     ......
     m_Progress.SetRange(0,100);              // 进程条的范围
     m_Progress.ShowWindow(SW_HIDE);          // 隐藏进程条

   m_Slider.SetRange(0,20);                   // 滑动条的范围
     m_Slider.ShowWindow(SW_HIDE);            // 隐藏滑动条
     SetTimer(100,500,NULL);                  //设置定时器
     return TRUE;   //return TRUE unless you set the focus to a control
}
```

### 7.6.4　操作滑动条

在 OnTimer()函数中添加、编辑如下代码：

```
void CMyPlayerDlg::OnTimer(UINT nIDEvent)
{
     CString strMode = GetMode();
```

```
        if(strMode.CompareNoCase("not open"))
        {
            if(IsMediumPresent())
            {
                ......
                char szPos[256];
                //操作滑动条
                m_Slider.SetPos((int)(20 * GetPosition(szPos)/GetLength())); m_dight
                = itoa((100 * GetPosition(szPos)/GetLength()),szPos,10);
                m_dight += "%";
                UpdateData(false);
            }
        else
        {
            GetDlgItem(IDC_BUTTON_PLAY)->EnableWindow(FALSE);
            GetDlgItem(IDC_BUTTON_STOP)->EnableWindow(FALSE);
        }
    }
    CDialog::OnTimer(nIDEvent);
}
```

### 7.6.5 编写响应滑动条操作的函数 OnHScroll()

使用“ClassWizard”来增加消息处理函数 OnHScroll 的步骤是：

（1）在“MFC ClassWizard”对话框中，进行如下设置：

```
Project: MyPlayer
Class name: CMyPlayerDlg
Object IDs: CMyPlayerDlg
Messager: WM_HSCROLL
```

（2）单击“Add Function”按钮，VC++便为滑动条的 WM_HSCROLL 事件添加了消息响应函数 OnHScroll()，如图 7.13 所示。

（3）点击“Edit Code”按钮，定位于 MplayerDlg.cpp 中的 OnHScroll()函数中，编辑如下函数代码：

```
void CMyPlayerDlg::OnHScroll(UINT nSBCode, UINT nPos, CScrollBar* pScrollBar)
{
    // TODO: Add your message handler code here and/or call default
    //get medium mode, if playing, stop
    CString strMode = GetMode();
    if(! strMode.CompareNoCase("playing"))
        Stop();
    // get current pos and seek it in medium
    int nSPos = m_Slider.GetPos();
    LONG lLength = GetLength();
    LONG lMPos = (LONG)((double)nSPos * lLength/20);
```

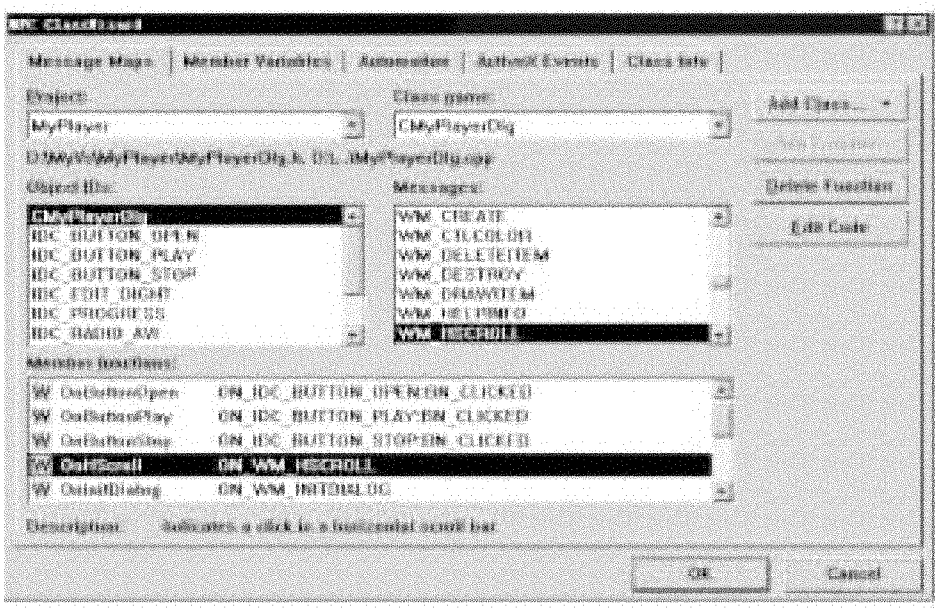


图 7.13　添加 WM _ HSCROLL 事件响应函数

```
    // normalize pos
    char szStartPos[256];
    lMPos = max(GetStartPosition(szStartPos), lMPos);
    lMPos = min(lLength, lMPos);
    // seek to lMPos
    Seek(lMPos);
    // if playing, play from lMPos
    if(! strMode.CompareNoCase("playing"))
        Play();

    CDialog::OnHScroll(nSBCode, nPos, pScrollBar);
}
```

### 7.6.6　构造并运行 MyPlayer

编译运行 MyPlayer 程序，先选取单选按钮，设置播放的媒体类型，再单击“打开”按钮，在“打开”对话框中，选取要播放的 * .midi 或 * .wave 或 * .avi 文件，在播放的过程中，用鼠标滑动滑动条控件，可跟踪媒体的播放位置，如图 7.1 所示。

### 7.6.7　技术要点

**1. Slider 控件**

Slider 控件由刻度和“滑块”共同构成，可由用户通过鼠标或箭头键来控制，以图形方式从一定的取值范围中选取一个数值。

可以在定义资源时对 Slider 控件的风格进行设置，具体见表 7.8。

表 7.8 Slider 控件 Styles 标签属性

| 选 项 | 说 明 |
|---|---|
| Orientation | 滑动条的方向可以是水平方向或者是垂直方向 |
| Point | 滑动条上的刻度位置<br>Both：两边都有<br>Top/Left：上边/左边有刻度<br>Bottom/Right：下边/右边有刻度 |
| Tick marks | 是否设置刻度 |
| Autoticks | 刻度自动进行设置 |
| Enable selection | 可以在滑动条上进行选择 |
| Border | 显示边框 |

Slider 控件与 CSliderCtrl 类相关联。CSliderCtrl 类的常用成员函数如表 7.9 所示。

表 7.9 CSliderCtrl 类的常用成员函数

| 函数原型 | 说 明 |
|---|---|
| void GetRange(int &nMin,int &nMax)const | 获取滑动条的数值范围 |
| void SetRange(int nMin,intnMax,BOOL bRedraw = FALSE) | 设置滑动条的数值范围 |
| void SetPos(int nPos) | 设置滑动条上当前的位置 |
| int GetPos()const | 获取滑动条上当前的位置 |

**2.** 滑动条的初始化

为了实现滑动条显示播放文件的进程，需要对滑动条初始化。函数 OnInitDialog()中的代码：

```
m_Slider.SetRange(0,20);
```

就是对滑动条进行初始化。

**3.** 操作滑动条

在定时器的响应函数 OnTimer()中的语句：

```
char szPos[256];
m_Slider.SetPos((int)(20 * GetPosition(szPos)/GetLength()));
```

用以更新滑动条的显示，其中 GetLength()是获取播放文件的长度，因此，数值(100 * GetPosition(szPos)/GetLength())为相对滑动条的最大值的相对数值。

## 习 题 七

1. 用发送命令消息的控制方式，修改 MyPlayer 程序。
2. 完善应用程序 MyPlayer，添加暂停、恢复等功能。
3. 完善应用程序 MyPlayer，添加录音功能。
4. 使用 MCIWND 窗口类，编写媒体播放器。

# 第8章

# 数据库编程

本章导读

Visual C++产品包含了两个相互独立的数据库访问系统：ODBC（开放式数据库连接）和DAO（数据访问对象），本章将介绍ODBC标准，利用MFC提供的ODBC类进行数据库编程，开发多媒体数据库管理程序和简易媒体点播系统，从而使读者掌握以下技术：

- 使用ODBC类进行数据库编程
- 自动注册数据源的ODBC数据库编程
- 在ODBC应用程序中使用SQL查询
- 媒体播放器与ODBC数据库编程的整合

---

## 8.1 简易媒体点播系统

### 8.1.1 简易媒体点播系统的功能

图8.1所示是一个简易媒体点播系统，具有以下功能：

（1）数据记录浏览：点击浏览记录的相应按钮，各编辑框中将显示当前记录。

（2）筛选节目：在编辑框中输入SQL语句，再点击“执行查询”按钮，则会关闭已打开的数据记录集，重新打开符合SQL查询条件的数据记录集，通过单击界面上的记录操作按钮可以验证查询结果。

（3）单击界面上的“播放”按钮，程序就会播放当前记录媒体记录（AVI、MIDI和三种媒体文件）。

### 8.1.2 点播系统开发步骤

本章首先介绍ODBC类的编程基础，再根据应用程序功能，结合知识点进行目标分解，分三步完成该系统的开发：

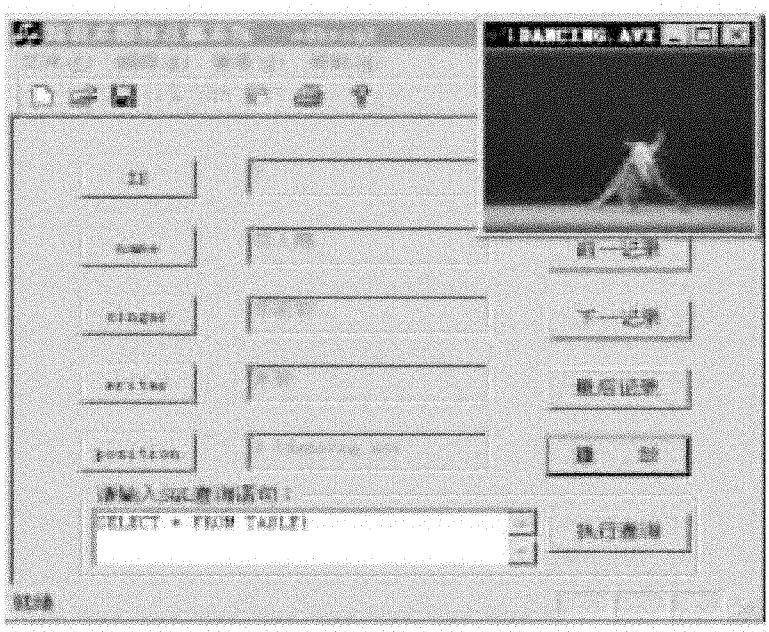


图 8.1 多媒体点播放系统

（1）利用 MFC 提供的 ODBC 类，开发一个简易多媒体数据库系统 odbc.exe，该系统具有数据记录的编辑和浏览功能。

（2）修改 odbc.exe，开发一个简易多媒体查询系统 odbcsql.exe，使之具有 SQL 查询功能。

（3）将第 7 章的媒体播放器与 odbcsql.exe 进行整合，完善 odbcsql 程序，编写一个简易媒体点播系统。

## 8.2 ODBC 类的编程基础

目前在各个领域应用的数据库种类较多，各种数据库的组成和管理也各不相同，所幸大部分数据库均支持一种标准接口：ODBC（开放式数据库连接），这种标准接口的一端可通过各种数据库驱动程序，访问数据库；另一端为应用程序和开发工具提供了一种统一的接口，也就是说，程序员只需要为 ODBC 操作编程，而不必关心 ODBC 是如何与数据库打交道的。

### 8.2.1 ODBC 的结构

ODBC 由 4 个部分组成，它们分别是：

（1）应用程序（Application）：负责处理和调用一系列的 ODBC 函数，用来提交 SQL 语句和检索结果。

（2）ODBC 管理器（ODBC Manage）：它是 4 个组成部分的核心，支持应用程序装载驱动程序，负责把用户的要求传递给 ODBC 驱动程序。

（3）ODBC 驱动程序（ODBC Drivers）：处理 ODBC 的调用，向特定数据源提交 SQL 请求，并向应用程序返回结果。

（4）数据源（Data Sources）：包含用户所要访问的数据，它关联着操作系统 DBMS，作为访问 DBMS 的网络平台。

ODBC 通过驱动程序来提供数据库的独立性。驱动程序和数据库有关，是支持 ODBC 函数调用的模块，应用程序通过调用驱动程序所支持的函数来操作数据库。应用程序通过动态连接到不同的驱动程序上来操作不同的数据库。驱动程序管理器还连接到所有的 ODBC 应用程序中，负责 ODBC 和 DLL 中函数的绑定。这种模式的关系如下所示：

应用程序↔ODBC 管理器↔ODBC 驱动程序↔数据库

### 8.2.2　MFC 提供的 ODBC 类

MFC 提供的 ODBC 类有：CDatabase、CRecordSet、CRecordView 和 CFilelExchange。

（1）CDatabase 负责连接数据源，它是针对某个数据库的，CDatabase 类对象提供了对数据库的连接，通过它可以对数据源进行操作。

（2）CRecordSet 针对数据源中的记录集，负责对记录的操作，CRecordSet 类对象提供了从数据源中提取出记录集的功能。CRecordSet 对象通常用于两种形式：动态行集（dynasets）和快照集（snapshots）。动态行集能保持与其他用户所做的更改保持同步。快照集则是数据集的一个静态视图。每一种形式在记录集被打开时都提供一组记录，所不同的是，当你在一个动态行集里滚动到一条记录时，由其他用户或是你应用程序中对该记录所做的更改会相应地显示出来。

（3）CRecordView 负责界面，CRecordView 类对象能以视图的形式显示数据库记录。这个视图是直接连接到一个 CRecordSet 对象的表视图。

（4）CFilelExchange 负责 CRecordSet 与数据源的数据交换。

### 8.2.3　应用 ODBC 编程

**1. ODBC 类编程的一般步骤**

使用 ODBC 类进行编程，一般步骤如下：

（1）连接数据源。

（2）创建并执行 SQL 语句。

（3）检查结果记录。

（4）断开数据源。

**2.** 数据记录的基本操作

假设定义子数据记录集：

```
CRecordSet *m_pSet;
m_pSet = new CRecordSet();
```

则将数据记录进行增加、删除和修改的方法如下：

（1）增加记录

使用 AddNew()函数增加记录，但要求数据库必须是以允许增加的方式打开，增加一条记录

的关键语句如下：

```
m_pSet->AddNew();                    //增加记录
......                               //输入新的字段值
m_pSet->Update();                    //将新记录存入数据库
m_pSet->Requery()                    //重建记录集
```

（2）删除记录

直接使用 Delete()函数，并且在调用 Delete()函数之后不需调用 Update()函数：

```
m_pSet->Delete();                    //删除记录
m_pSet->Requery();                   //重建记录集
```

（3）修改记录

修改记录使用 Edit()函数，关键语句如下：

```
m_pSet->Edit();                      //修改当前记录
......                               //修改当前记录字段值
m_pSet->Update();                    //将新记录存入数据库
m_pSet->Requery()                    //重建记录集
```

### 8.2.4 创建数据源（DSN）

前面提到过 ODBC 是一个方便的接口，它和具体用哪种数据库系统创建的数据库没关系，但要访问数据库，就必须为数据库指定一个惟一的名称，即添加 ODBC 数据源，否则应用程序将无法访问。

假定用户已创建了一个名为 myplayer.mdb 的 Access 数据库，并存放在 D:\MyVc 目录下。则创建 ODBC 数据源的步骤如下：

（1）在控制面板中，双击“32bits 数据源”图标打开“ODBC 数据源管理器”（对于 Windows 2000 环境，在控制面板中，需先打开“管理工具”，才能找到数据源图标），并选择用户 DSN（即 Data Source），如图 8.2 所示。可以看到在用户数据源里有很多的驱动程序列表，在此向里面添加用户

图 8.2 “ODBC 数据源管理器”对话框

所需要的数据源。

（2）单击“添加(D)...”按钮，显示“创建新数据源”对话框，选择“Drive do Microsoft Access(*mdb)”，添加 Microsoft Access 驱动程序，如图 8.3 所示。

图 8.3　“创建新数据源”对话框

（3）单击“完成”按钮，在弹出的“ODBC Microsoft Access 安装”对话框(如图 8.5 所示)中，输入要添加的数据源名，并加入必要的说明，点击“选取”按钮，会弹出“选定数据库”对话框，选择存放 myplayer.mdb 文件的驱动器和目录，再选中 myplayer.mdb 文件，如图 8.4 所示。

图 8.4　“选定数据库”对话框

（4）点击“确定”，返回“ODBC Microsoft Access 安装”对话框，如图 8.5 所示。

（5）点击“确定”按钮，返回到“ODBC 数据源管理器”，新添加的数据源显示在用户数据源列表中，如图 8.6 所示。

（6）点击“确定”按钮，就完成了一个 ODBC 数据源的添加工作。

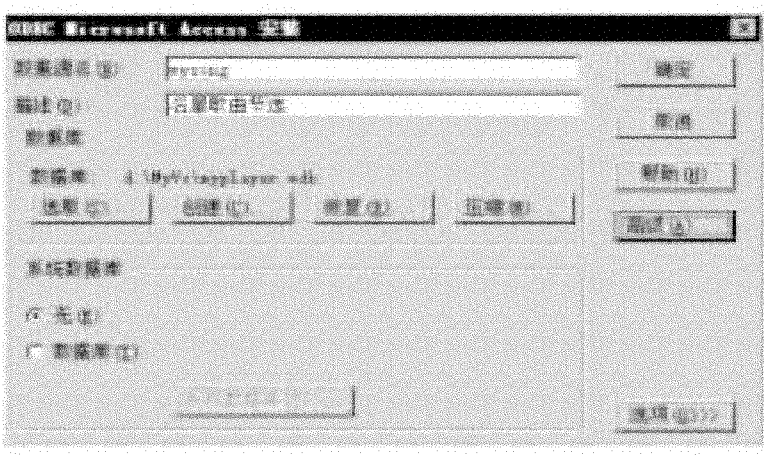


图 8.5 “ODBC Microsoft Access 安装”对话框

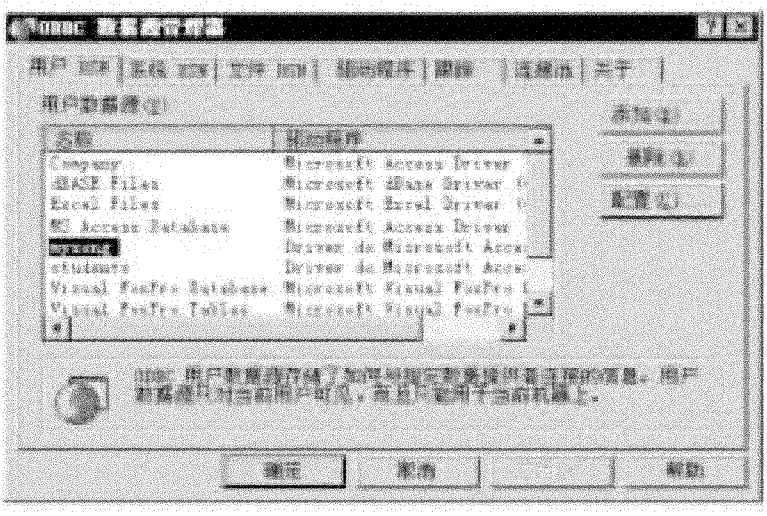


图 8.6 添加数据源后的“ODBC 数据源管理器”

### 8.2.5 在 ODBC 应用程序中注册数据源

通常，在使用 ODBC 之前，用户需要启动 ODBC 数据源管理器，对数据源进行注册，这对于用户而言实在麻烦，下面介绍在 ODBC 应用程序中注册数据源的方法。

在 ODBC 应用程序中注册数据源打开记录集的步骤如图 8.7 所示。

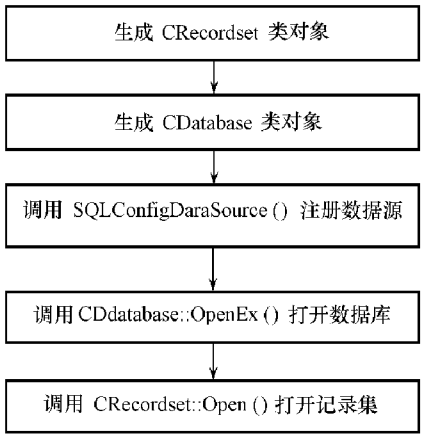



图 8.7 注册数据源的步骤

在 ODBC 应用程序中注册数据源的核心是调用::SQLConfigDataSource( )函数。该函数原型为：

```
BOOL SQLConfigDataSource(
        HWND hwndParent,
        WORD fRequest,
```

```
                LOCSTR lpszDriver,
                LOCSTR lpszAttributes);
```

例如，注册一个数据源为 msong，且数据源文件绝对路经为 D:\ \ MyVC \ \ msong.mdb 的 Microsoft Access 数据源。调用方法为：

```
SQLConfigDataSource(NULL,                                    //不需要弹出对话框
                    ODBC_ADD_DSN,                            //添加数据源
                    "Microsoft Access Driver(*.mdb)",        //驱动程序名
                    "DSN=msong\0"                            //数据源名称
                    "Description=This is a sample\0"         //数据源的说明
                    "FileType= Microsoft Access\0"           //数据源文件类型说明
                    "DBQ=D:\ \MyVC\ \msong.mdb\0"            //数据源文件绝对路经
                    "MaxScanRows=8\0"                        //在根据现有数据设置列的数
                                                             //据类型时所要扫描的行数
                    //可以为 1 到 16,默认值为 8;如果设置为 0
                    //将扫描所有行。如果数字超出界限,会返回一个错误
                  );
```

## 8.3　多媒体数据库

本节将利用 MFC 提供的 ODBC 类，开发一个简易多媒体数据库程序，该程序具有数据记录的编辑和浏览功能，如图 8.8 所示。

图 8.8　多媒体数据库程序(odbc.exe)运行界面

### 8.3.1 创建工程

创建一个基于单文档界面(SDI)的 odbc 程序,创建工程的步骤如下:

(1) 启动 Visual C++6.0。从"File"菜单中选择"New"。在"New"对话框中选择"Project"标签。

(2) 在"New"对话框的"Project name"文本编辑框中输入"odbc",单击位于"Location"框右边的小按钮,再从下拉的对话框中选择"d:\MYVC"目录,使新创建的工程文件放置在"d:\MYVC"目录之下。

(3) 单击"New"对话框的"OK"按钮。此时 Visual C++将显示"MFC AppWizard - Step 1"对话框。

(4) 在"MFC AppWizard - Step 1"对话框中,选择"Single document",即创建一个基于对话框的应用程序,然后单击"Next"按钮。

(5) 在"MFC AppWizard - Step 2 of 6"对话框中,选择"Database view without file support"单选项,如图 8.9 所示。

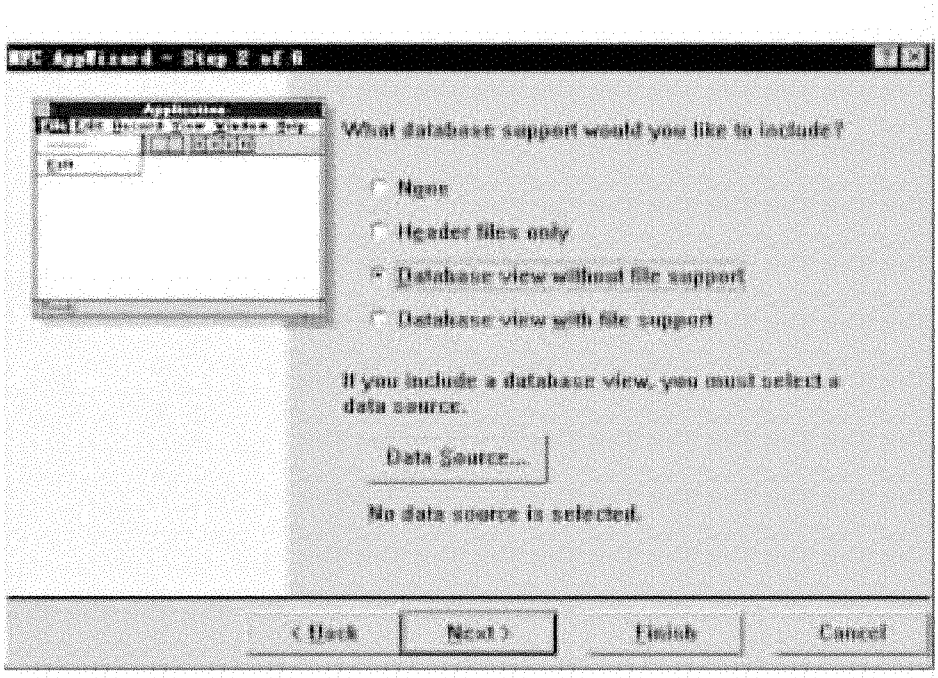



图 8.9 设置数据库支持

(6) 单击"Data Source..."按钮,弹出"Database Options"对话框。在"Database Options"对话框中,选取下拉式列表框中的"mysong"数据源(8.2.4 节创建的数据源),如图 8.10 所示。

(7) 单击"OK"按钮,将弹出"Select Database Tables"对话框。在"Select Database Tables"对话框中,将列出 mysong 数据源中的所有数据表,选择要操作的数据表"Table Song",如图 8.11 所示。

(8) 单击"OK"按钮。此时 Visual C++将返回到图 8.9 的"MFC AppWizard - Step 2 of 6"对话框,至此,数据源设置完毕。

(9) 单击"Next"按钮。在后续的各向导对话框中保留其默认设置(在此略),Visual C++就会创建支持数据库操作的 odbc 工程以及相关的所有文件。

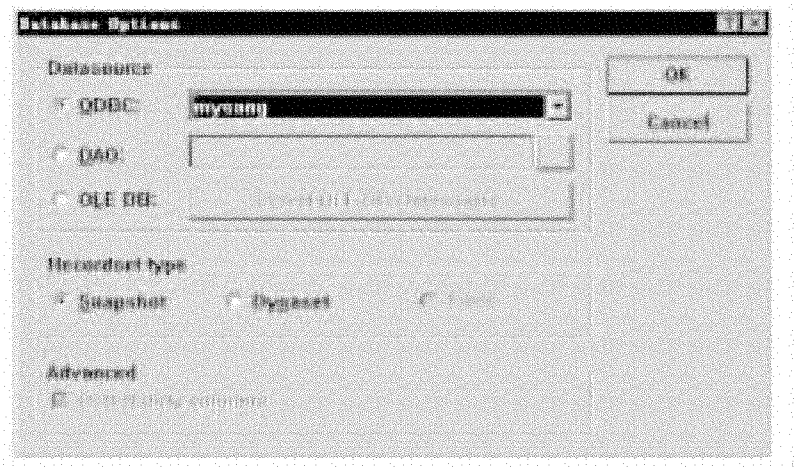



图 8.10　“Database Options”对话框

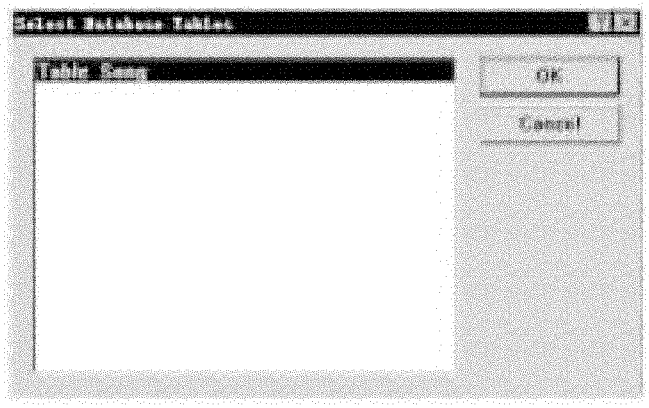



图 8.11　“Select Database Tables”对话框

### 8.3.2　可视化设计

设计主窗口界面的步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 odbc resources 扩展开，然后再扩展 Dialog，最后，双击“IDD _ ODBC _ FORM”项，Visual C ++ 显示出处于设计状态的“IDD _ ODBC _ FORM”对话框。

（2）在“IDD _ ODBC _ FORM”对话框中，根据表 8.1 中的定义编辑对话框资源，设计完毕的对话框如图 8.12 所示。

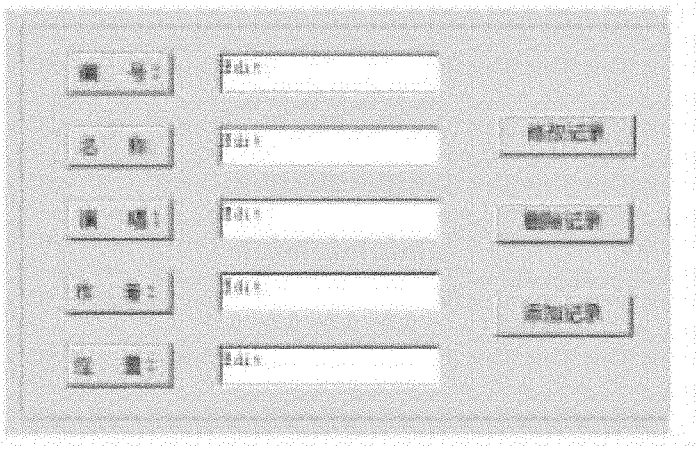


图 8.12　设计完后的“IDD _ ODBC _ FORM”对话框

表 8.1 “IDD _ ODBC _ FORM”对话框中各控件属性

| 对 象 | 属 性 | 属性值 |
|---|---|---|
| Edit Box | ID，Border | IDC _ EDIT _ ID，Checked( Styles) |
| Edit Box | ID，Border | IDC __ EDIT _ NAME，Checked( Styles) |
| Edit Box | ID，Border | IDC __ EDIT _ SINGER ，Checked( Styles) |
| Edit Box | ID，Border | IDC __ EDIT _ WRITER，Checked( Styles) |
| Edit Box | ID，Border | IDC __ EDIT _ POSITION，Checked( Styles) |
| Static Text | ID，Caption<br>Center Vertically<br>Modal frame | STATIC，编号：<br>Checked(Styles)<br>Checked(Extended Styles) |
| | | |
| Static Text | ID，Caption<br>Center Vertically<br>Modal frame | STATIC，位置：<br>Checked(Styles)<br>Checked(Extended Styles) |
| Button | ID，Caption | IDR _ RECORD _ ADD，添加记录 |
| Button | ID，Caption | IDR _ RECORD _ DEL，删除记录 |
| Button | ID，Caption | IDR _ RECORD _ EDIT，修改记录 |

### 8.3.3 为各 Edit Box 编辑框引入变量

为了在浏览数据记录时，各编辑框能显示当前记录对应的字段，必须为它们引入变量。

为编辑控件引入变量的操作步骤如下：

（1）选择“View”菜单中的“ClassWizard”菜单项。

（2）在“MFC ClassWizard”中，选择“Member Variables”标签，并进行如下设置：

```
Class name: COdbcView
Control IDs: IDC _ EDIT _ ID
```

（3）单击“Add Variable...”按钮，弹出“Add Member Variable”对话框，进行如下设置：

```
Variable name: m _ ID
Category: Value
Variable type: UINT
```

（4）单击“OK”按钮，此时“MFC ClassWizard”就为 IDC _ EDIT _ ID 编辑框引入变量 m _ ID。

（5）完全类似地，重复(2)~(4)步的方法，为 Edit Box 编辑框：IDC _ EDIT _ NAME、IDC _ EDIT _ SINGER、IDC _ EDIT _ WRITER 和 IDC _ EDIT _ POSITION 引入变量，如图 8.13 所示。

（6）单击“MFC ClassWizard”对话框中的“OK”按钮，就为“IDD _ ODBC _ FORM”对话框中的各 Edit Box 编辑框引入变了量。

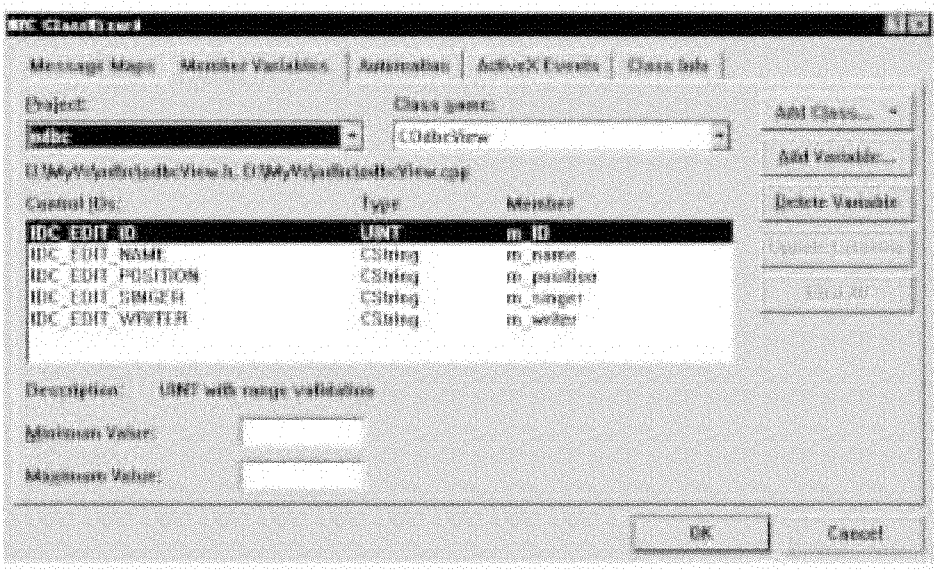



图 8.13　为各 Edit Box 编辑框引入关联变量

### 8.3.4　添加消息响应函数

**1.** 为“记录(**R**)”菜单下的各菜单项添加消息响应函数

操作步骤如下：

(1) 从“View”菜单中选择“ClassWizard”菜单项，弹出“MFC ClassWizard”对话框。

(2) 选择“Message Maps”标签，并进行如下设置：

```
Class name: COdbcView
Object IDs: IDC_RECORD_FIRST
Messages: COMMAND
```

(3) 单击“Add Function”按钮来增加新函数，在弹出的“Add Member Function”对话框中，接受默认函数名 OnRecordFirst。

(4) 单击“OK”按钮，就为“第一个记录(F)”菜单项添加消息响应函数 OnRecordFirst()。同时也就为工具条上记录移动的对应按钮“IDC_RECORD_FIRST”添加了消息响应函数。因为它们有相同的 ID 标识：IDC_RECORD_FIRST。

(5) 完全类似地，重复(2)~(4)步的方法，如图 8.14 所示，为“记录(R)”菜单下的其他记录移动菜单项添加消息响应函数：OnRecordLast()、OnRecordNext()和 OnRecordPrev()。

(6) 单击“MFC ClassWizard”对话框中的“OK”按钮，为“记录(R)”菜单下的各菜单项添加消息响应函数。

**2.** 为“添加记录”按钮的 **BN_CLICKED** 事件添加消息响应函数

操作步骤如下：

(1) 从“View”菜单中选择“ClassWizard”菜单项，弹出“MFC ClassWizard”对话框。

(2) 选择“Message Maps”标签，并进行如下设置：

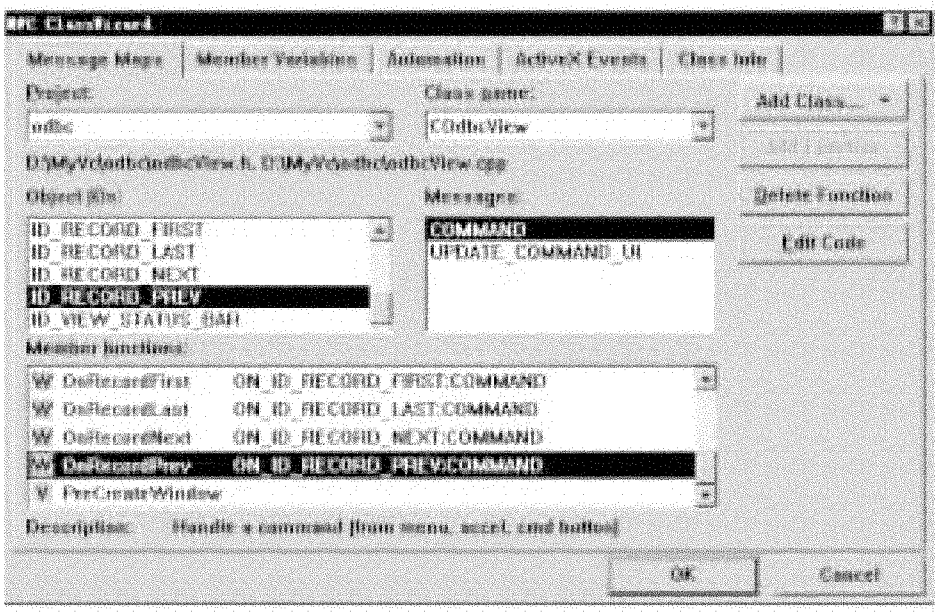



图 8.14 添加消息响应函数后的“MFC ClassWizard”对话框

```
Class name: COdbcView
Object IDs: IDC_RECORD_ADD
Messages: BN_CLICKED
```

(3) 单击“Add Function”按钮来增加新函数,在弹出的“Add Member Function”对话框中,接受默认函数名 OnRecordAdd。

(4) 单击“OK”按钮。此时,就为“添加记录”按钮“IDC_RECORD_ADD”添加消息响应函数 OnRecordAdd()。

完全类似地,重复(2)~(4)步的方法,如图 8.15 所示,为“删除记录”按钮 IDR_RECORD_DEL 和“修改记录”按钮 IDR_RECORD_EDIT 添加消息响应函数 OnRecordDel()和 OnRecordEdit()。

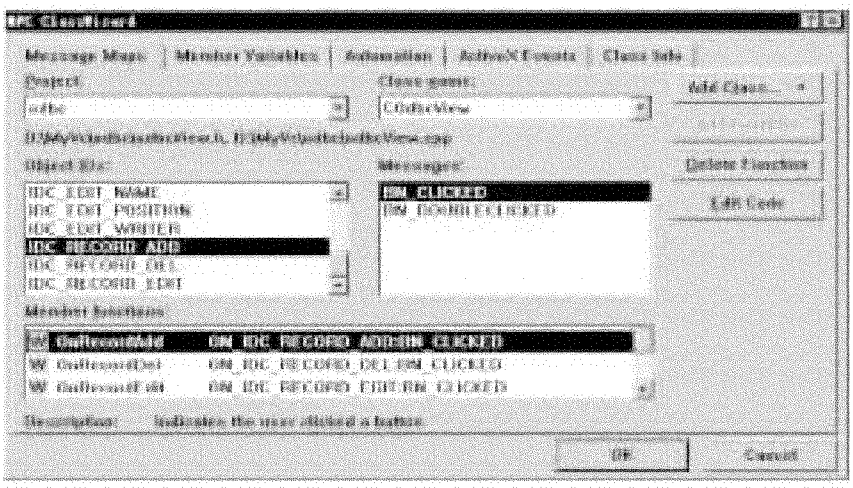



图 8.15 添加消息响应函数后的“MFC ClassWizard”对话框

### 8.3.5　编写程序代码

**1. 在 COdbcView 类中添加自定义函数**

(1) 自定义函数原型声明

打开 odbcView.h 文件，在 COdbcView 类的定义中添加自定义函数原型：

```
void MyUpdateData(void);
class COdbcView : public CRecordView
{
protected:  // create from serialization only
    COdbcView();
    DECLARE_DYNCREATE(COdbcView)
    ......
// Attributes
public:
    COdbcDoc* GetDocument();
    void MyUpdateData(void);// 自定义函数原型说明
// Operations
public:
    ......
};
```

(2) 添加自定义函数体

该函数的功能就是从打开的记录集中，将当前记录指针所指的记录传递给主窗口对话框中的各编辑框的关联变量，并更新屏幕。从而，使当前记录指针所指的记录显示在各编辑框中。

打开 odbcView.cpp 文件，在程序末尾，添加如下自定义函数体：

```
void COdbcView:: MyUpdateData()
{
   m_name = m_pSet->m_name;
   m_ID = m_pSet->m_ID;
   m_singer = m_pSet->m_singer;
   m_writer = m_pSet->m_writer;
   m_position = m_pSet->m_position;
   UpdateData(false);
}
```

**2. 为“记录(R)”菜单下的各菜单项的消息响应函数编写代码**

为各记录移动消息响应函数添加如下代码：

```
void COdbcView::OnRecordFirst()
{
   m_pSet->MoveFirst();      //移动记录集指针至第一条记录处
   MyUpdateData();           //更新记录显示
}
```

```
void COdbcView::OnRecordLast()
{
   m_pSet->MoveLast();          //移动记录集指针至最后一条记录处
   MyUpdateData();              //更新记录显示
}

void COdbcView::OnRecordNext()
{
   m_pSet->MoveNext();          //移动记录集指针至下一条记录处
   MyUpdateData();              //更新记录显示
}

void COdbcView::OnRecordPrev()
{
   m_pSet->MovePrev();          //移动记录集指针至前一条记录处
   MyUpdateData();              //更新记录显示
}
```

**3.** 修改视类成员函数 **OnInitialUpdate()**

在 OnInitialUpdate()中添加创建并打开数据集代码：

```
void COdbcView::OnInitialUpdate()
{
   m_pSet = &GetDocument()->m_odbcSet;
   CRecordView::OnInitialUpdate();
   GetParentFrame()->RecalcLayout();
   ResizeParentToFit();
   //创建并打开数据集
   try
   {
      m_pSet = new COdbcSet();
      m_pSet->Open();
      MyUpdateData();
   }
   catch(CDBException* pe)
   {
      AfxMessageBox(pe->m_strError);
      pe->Delete();
   }
}
```

**4.** 为函数 **OnRecordAdd()**、**OnRecordDel()**和 **OnRecordEdit()**编写代码

将各编辑框的内容传递给各关联变量，再将各变量传递给数据集指针所指记录的各字段，更新记录集：

```
void COdbcView::OnRecordAdd()
{
    try
```

```
        {
            m_pSet->AddNew();
            // 将各编辑框的内容传递给各关联变量
            UpdateData(true);
            //将各变量传递给数据集指针所指记录各字段
            m_pSet->m_ID = m_ID;
            m_pSet->m_name = m_name;
            m_pSet->m_singer = m_singer;
            m_pSet->m_writer = m_writer;
            m_pSet->m_position = m_position;
            m_pSet->Update();
            m_pSet->Requery();
        }
        catch(CDBException* pe)
        {
            AfxMessageBox(pe->m_strError);
            pe->Delete();
        }
}

void COdbcView::OnRecordDel()
{
        try
        {
            m_pSet->Delete();
            m_pSet->Requery();
            MyUpdateData();
        }
        catch(CDBException* pe)
        {
            AfxMessageBox(pe->m_strError);
            pe->Delete();
        }
}

void COdbcView::OnRecordEdit()
{
        try
        {
            m_pSet->Edit();
            UpdateData(true);
            m_pSet->m_ID = m_ID;
            m_pSet->m_name = m_name;
            m_pSet->m_singer = m_singer;
            m_pSet->m_writer = m_writer;
            m_pSet->m_position = m_position;
            m_pSet->Update();
        }
```

```
    catch(CDBException* pe)
    {
      AfxMessageBox(pe->m_strError);
      pe->Delete();
    }
}
```

### 8.3.6 查看结果

按如下步骤操作,观察结果。

(1) 编辑运行 odbc 程序,出现图 8.8 所示的运行界面。

(2) 单击工具栏上的记录移动按钮或点击相应菜单项,各 Edit Box 框内将显示对应的记录数据。

(3) 编辑各字段数据,单击"修改记录"或"添加记录"按钮,将完成相应功能。

(4) 单击"删除记录"按钮,界面上显示的当前记录就被删除。

### 8.3.7 技术要点

**1.** 与数据库的连接

本例中的 COdbcSet 类是 CRecordSet 的派生类,这个类对于连接数据库操作非常重要。经 ODBC 访问数据库的操作通过成员函数 GetDefaultConnect()实现,本例中开始设置的数据库源连接体现在 OdbcSet.cpp 的如下代码中:

```
CString COdbcSet::GetDefaultConnect()
{
    return _T("ODBC;DSN=mysong");
}

CString COdbcSet::GetDefaultSQL()
{
    return _T("[Table1]");
}
```

**2.** 当前数据记录与 **COdbcSet** 类成员变量的数据交互

另外,COdbcSet 类中的 DoFieldExchange()成员函数完成数据库一条记录的数据与 COdbcSet 类中相应成员变量的数据交互,从而可以通过操作这些成员变量完成操作数据库。代码如下:

```
void COdbcSet::DoFieldExchange(CFieldExchange* pFX)
{
  //{{AFX_FIELD_MAP(COdbcSet)
  pFX->SetFieldType(CFieldExchange::outputColumn);
  RFX_Long(pFX, _T("[ID]"), m_ID);
  RFX_Text(pFX, _T("[name]"), m_name);
  RFX_Text(pFX, _T("[singer]"), m_singer);
  RFX_Text(pFX, _T("[writer]"), m_writer);
```

```
    RFX_Text(pFX, _T("[position]"), m_position);
    //}}AFX_FIELD_MAP
}
```

### 3. RecordSet 类简介

RecordSet 类派生于 CObject 类。一个 CObject 类对象代表了从数据源中选出的一组记录，即记录集。利用这个类，可以将编程逻辑同实际的 SQL 相隔离，SQL 对数据库所做的 SELECT、INSERT、UPDATE 等操作都可以用该类来实现。从这个类派生的类也都具有这些特性。

使用该类的方法是从该类派生出一个新类，记录集会从数据源选择数据集，然后用户就可以进行如下操作：

（1）在记录集中移动。

（2）更新记录同时指定一个锁定模式。

（3）从记录集中过滤出需要的记录。

（4）参数化查询条件，直到在运行时间才确定具体内容。

RecordSet 类的常用成员函数如表 8.2 所示。

表 8.2　RecordSet 类常用成员函数

| 函数名称 | 作　用 |
|---|---|
| Open | 通过返回一个表或一次查询打开一个记录集 |
| Close | 关闭数据集 |
| IsBOF | 判断记录集是否指向第一条记录之前，即是否为空 |
| IsEOF | 判断记录集是否指向最后一条记录之后，即是否为空 |
| IsDeleted | 判断记录集是否指向了一条被删除的记录 |
| IsOpen | 看数据库是否被打开（是否调用过 Open） |
| AddNew | 准备添加新记录，要调用 Update 来完成添加 |
| CancelUpdate | 取消通过 AddNew 或 Edit 操作进行的数据改变 |
| Delete | 删除当前记录，完成后要移动指针到另外的位置 |
| Edit | 准备改变当前记录，要调用 Update 来完成 |
| Update | 用来完成 AddNew 或 Edit 操作 |
| Move | 将记录集定位到指定的位置 |
| MoveFirst | 将记录集定位到首记录，用以测定 IsBOF |
| MoveLast | 将记录集定位到末记录，用以测定 IsEOF |
| MoveNext | 移动到下一条记录 |
| MovePrev | 移动到上一条记录 |
| SetAbsolutePosition | 定位记录集到指定的位置 |
| GetDefaultConnect | 获得一个默认的连接 |
| GetDefaultSQL | 调用一个默认的 SQL 语句串执行 |
| DoFieldExchange | 进行数据域交换 |

# 8.4 多媒体查询系统

前面开发的多媒体数据库 odbc.exe，在利用 MFC 进行数据显示时，其做法是生成一个 CRecordset 类的派生类，然后利用 CRecordset 类的派生类，通过 RFX(Record Field Exchange)技术获取记录集数据。

然而，在 CRecordView 类显示数据时，使用 CRecordset 类的派生类虽然可以少编甚至不编代码，但是却丧失了随心所欲查看、修改数据库中各种数据的灵活性。

为此，本节将利用 8.2.5 自动注册数据源和 SQL 查询语句，开发一个简易多媒体查询系统，随心所欲查看数据，如图 8.16 所示。


图 8.16 简易多媒体查询系统

与 ODBC.EXE 程序相比，有如下改进：

- 避开必须先注册数据源，在创建应用程序框架时选定数据源等限制，而是在应用程序中自动注册数据源，更具灵活性和适用性。
- 添加了 SQL 查询功能，具有随心所欲查看、修改数据库中各种数据的灵活性。

## 8.4.1 SQL 查询简介

ODBC 类对 SQL 语句的支持，使用户可以随心所欲查看、修改数据。

**1. SQL 查询语句**

结构化查询语句 SQL 是处理关系数据库的专门语言。通过 SQL 语言，用户可以按照一定的约束条件获取一组记录，然后对它们进行各种操作，诸如：查询、修改、删除和添加等。常用的

SQL查询语句如表8.3所示。各种SQL语句的用法举例，请读者参考有关书籍，在此从略。

表8.3　常用的SQL查询语句

| SQL语句 | 语　法 | 作　用 |
|---|---|---|
| SELECT | SELECT 要检索的字段 FROM 要检索的表 | 检索数据库中的数据 |
| ORDER BY | SELECT 要检索的字段 FROM 要检索的表 ORDER BY 某指定字段 | 对检索结果进行排序 |
| WHERE | SELECT * FROM 要检索的表 WHERE 指定字段 = 特定值 | 仅提取某些满足条件的记录 |
| INNER JOIN | FROM table1 INNER JOIN table2 ON table1.field1 compopr table2.fielf2 | 在table1和table2中有相同字段时，将两个表中的记录组合在一起显示 |
| UPDATE | UPDATE 表名 SET 字段名 = 新值 | 用于更新记录 |

**2. SQL查询方法**

显然，在应用程序中执行SQL语句，可以极大地增强应用程序的灵活性和功能，而CDatabase类和Crecordset类对此有很好的支持。

在ODBC应用程序中实现对数据的任意查询，其操作步骤是：

（1）首先必须关闭记录集（如果已打开）

```
if(m_set->IsOpen())m_set->Close();
```

（2）调用CRecordset::Open()函数打开记录集并指定SQL查询语句

例如，若要查看所有年龄小于26岁的作者的信息，则用如下语句打开记录集：

```
m_set->Open(CRecordset::dynaset,
                _T("SELECT * FROM TABLE1 WHERE age<=26"));
```

（3）执行一般的查询语句

CDatabase类提供了一个名为ExecuteSQL()的函数，利用该函数可以进行SQL调用，CDatabase::ExecuteSQL()函数的原型定义如下：

```
void ExecuteSQL(LPCSTR lpszSQL);throw(CDBException);
```

其中参数lpszSQL代表需要执行的SQL语句。

例如：

```
CDatabase m_Db;
CString m_Statement = "UPDATE Table1 SET name=New Value";
TRY {
    m_Db.ExecuteSQL(m_Statement);
}
CATCH(CDBException, e)
{
   ......  //进行异常处理
```

}

### 8.4.2 创建工程

利用“MFC ClassWizard”创建一个基于单文档的工程，工程取 odbcsql。要特别注意的是：在“MFC AppWizard – Step 6 of 6”对话框中，选取 CFormView 作为视类的基类，其目的是在视图类增加对话框，用于添加控件，操作与显示数据，如图 8.17 所示。其他每一步均接受默认设置。

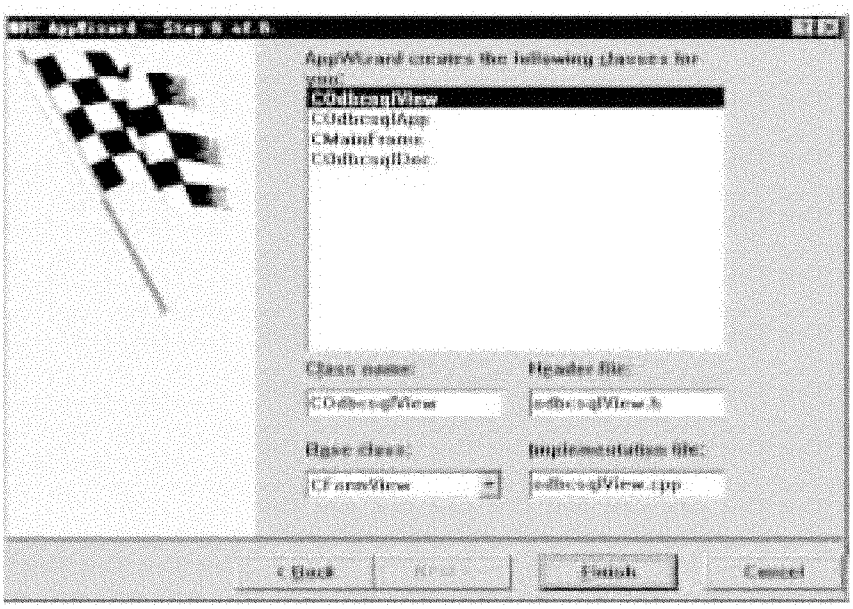


图 8.17 “MFC AppWizard – Step 6 of 6”对话框

### 8.4.3 可视化设计

可视化设计主窗口界面的步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 odbcsql resources 扩展开，然后再扩展 Dialog，最后，双击“IDD _ ODBCSQL _ FORM”项，Visual C++ 显示出处于设计状态的“IDD _ ODBCSQL _ FORM”对话框。

（2）在“IDD _ ODBCSQL _ FORM”对话框中，根据表 8.4 中的控件属性编辑对话框资源，设计完毕后对话框如图 8.18 所示。

表 **8.4** “**IDD _ ODBCSQL _ FORM**”对话框中各控件属性表

| 对 象 | 属 性 | 属性值 |
|---|---|---|
| Edit Box | ID<br>Border，Read-Only | IDC _ EDIT _ ID<br>Checked（Styles tab） |
| Edit Box | ID<br>Border，Read-Only | IDC _ EDIT _ NAME<br>Checked（Styles tab） |

（续表）

| 对　象 | 属　性 | 属性值 |
| --- | --- | --- |
| Edit　Box | ID<br>Border，Read-Only | IDC _ EDIT _ SINGER<br>Checked（ Styles tab） |
| Edit　Box | ID<br>Border，Read-Only | IDC _ EDIT _ WRITER<br>Checked（ Styles tab） |
| Edit　Box | ID<br>Border，Read-Only | IDC _ EDIT _ POSITION<br>Checked（ Styles tab） |
| Static Text | ID，Caption<br>Center Vertically<br>Modal frame | IDC _ ITEM1，字段 1<br>Checked（Styles）<br>Checked（Extended Styles） |
| | | |
| Static Text | ID，Caption<br>Center Vertically<br>Modal frame | IDC _ ITEM5，字段 5<br>Checked（Styles）<br>Checked（Extended Styles） |
| Button | ID，Caption | IDC _ RECORD _ FIRST，第一记录 |
| Button | ID，Caption | IDC _ RECORD _ PREV，前一记录 |
| Button | ID，Caption | IDC _ RECORD _ NEXT，下一记录 |
| Button | ID，Caption | IDC _ RECORD _ LAST，最后记录 |
| Button | ID，Caption | IDC _ SQL _ OK，执行查询 |
| Edit　Box | ID<br>Multiline，Vertical scroll<br>Auto VScroll，Uppercase | IDC _ EDIT _ SQL<br>Checked（Styles tab）<br>Checked（Styles tab） |

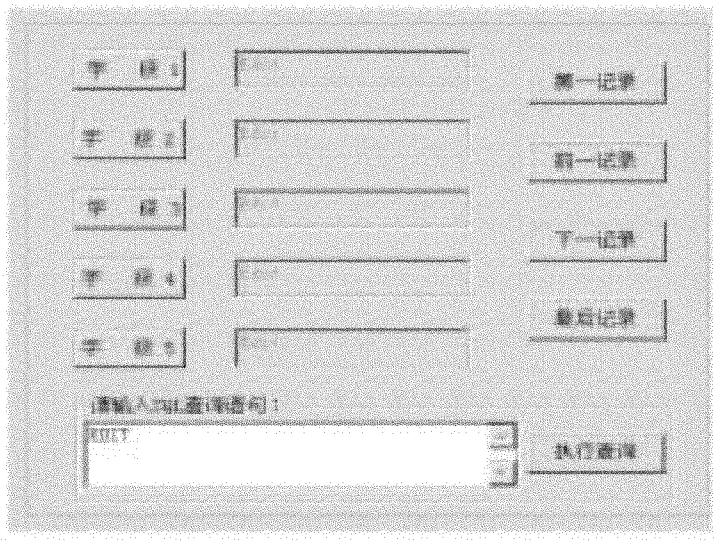


图 8.18　设计好的主界面

### 8.4.4 给各控件引入变量

用“MFC ClassWizard”为“IDD _ ODBCSQL _ FORM”对话框中的 5 个 Edit Box 控件(IDC _ EDIT _ ID、IDC _ EDIT _ NAME、IDC _ EDIT _ SINGER、IDC _ EDIT _ WRITER 和 IDC _ EDIT _ POSITION)和 5 个 Static Text 控件(IDC _ ITEM1、IDC _ ITEM2、IDC _ ITEM3、IDC _ ITEM4 和 IDC _ ITEM5)引入关联变量,如图 8.19 所示。

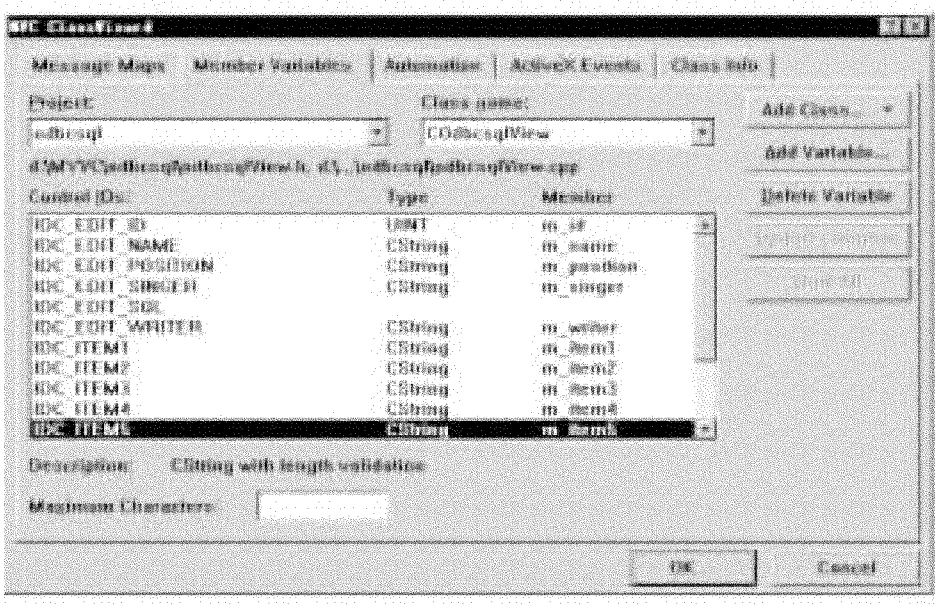



图 8.19 给“IDD _ ODBCSQL _ FORM”对话框中各控件引入关联变量

### 8.4.5 修改视图类 COdbcsqlView

为了在视图类操作和显示数据记录,必须在 COdbcsqlView 类添加数据成员和函数,并添加支持数据库所需头文件和自动注册数据源库函数所需的头文件。

**1.** 添加支持数据库的头文件

在 odbcsqlView.h 文件中,添加支持数据库所需头文件和自动注册数据源库函数所需的头文件:

```
#include <afxdb.h>          //数据库支持所需头文件
#include"ODBCINST.H"        //SQLConfigDataSource()函数所在的头文件
```

**2.** 为 **COdbcsqlView** 类添加数据成员变量

在 COdbcsqlView 类的定义中,添加用于对数据库操作的变量。

```
class COdbcsqlView : public CFormView
{
protected: // create from serialization only
        COdbcsqlView();
```

```
        DECLARE_DYNCREATE(COdbcsqlView)
        //添加数据成员
        short m_NKey;
        CString m_Key;
        CRecordset* m_set;
        int m_CurSel;
        ......
public:
        CString m_Query;
        BOOL m_DSOK;
        CDatabase m_Db;
        ......
        DECLARE_MESSAGE_MAP()
};
```

**3.** 初始化数据成员变量

在构造函数中初始化成员变量：

```
COdbcsqlView::COdbcsqlView()
        : CFormView(COdbcsqlView::IDD)
{
       //{{AFX_DATA_INIT(COdbcsqlView)
        ......
        //}}AFX_DATA_INIT
        m_set = NULL;
        m_NKey = 0;
        m_DSOK = TRUE;
        m_Query = "Select * from Table1";
}
```

**4.** 添加函数成员

在 COdbcsqlView 类的公有段中(odbcsqlView.h 文件中)添加函数成员：

```
void ReadDisplayFields(void);
void ReadDispalyRecord(void);
```

并在 odbcsqlView.cpp 文件中添加方法的实现函数，以便将当前记录显示在对应的 static Text 和 Edit Box 框中：

```
void COdbcsqlView::ReadDisplayFields(void)
{  short i, nFields;
         CODBCFieldInfo Fi;
         nFields = m_set->GetODBCFieldCount();
         for(i = 0;i < nFields;i++)
         {  m_set->GetODBCFieldInfo(i,Fi);
            if(i == 0)m_item1 = Fi.m_strName;
            if(i == 1)m_item2 = Fi.m_strName;
            if(i == 2)m_item3 = Fi.m_strName;
```

```
            if(i==3)m_item4= Fi.m_strName;
            if(i==4)m_item5= Fi.m_strName;
    }
}

void COdbcsqlView::ReadDispalyRecord(void)
{
    if(m_set->IsEOF()==0)
  { short j, nFields;
        CString str;
        nFields=m_set->GetODBCFieldCount();
        for(j=0;j<nFields;j++)
        {
            m_set->GetFieldValue((short)j,str);  //获取数据
            if(j==0)m_id = atol(str);
            if(j==1)m_name = str;
            if(j==2)m_singer = str;
            if(j==3)m_writer=str;
            if(j==4)m_position =str;
        }
    }
}
```

### 8.4.6 修改 OnInitialUpdate()函数

要操作数据库，首先必须在 COdbcsqlView::OnInitialUpdate()函数中注册数据源，然后建立同数据库的连接，打开记录集：

```
void COdbcsqlView::OnInitialUpdate()
{  CFormView::OnInitialUpdate();
   GetParentFrame()->RecalcLayout();
   ResizeParentToFit();
   //1.注册数据源
   if(m_set==NULL)
   {  if(! SQLConfigDataSource(
      NULL,                                    //父窗口指针
      ODBC_ADD_DSN,                            //请求的类型
      "Microsoft Access Driver(*.mdb)",        //驱动程序名;属性
      "DSN=msong\0"                            //数据源名称
      "Description=dbdata database\0"          //数据源的说明
      "FileType=Microsoft Access\0"            //数据源文件类型说明
      "DBQ=D:\\MyVc\\song.mdb\0"               //数据源文件全路径名
      "MaxScanRows=8\0"  //在根据现有数据设置列的数据类型时所要扫描的行数
                         //可以为 1 到 16,默认值为 8;如果设置为 0
                         //将扫描所有行。如果数字超出界限,会返回一个错误
      ))
   {
```

```
        AfxMessageBox("无法创建数据源!");
        m_DSCK = FALSE;
        return;
    }
    m_set = new CRecordset(&m_Db);
    if(! m_Db.OpenEx(_T("DSN = msong"),0))    //建立同数据库的连接
    {
        AfxMessageBox("你选择了取消");
        return;
    }
    m_set->Open( CRecordset::dynaset, _T(m_Query));  // 打开记录集
    //数据源注册完毕
    //2. 读出并显示数据库字段名
    ReadDisplayFields();
    //3. 添加行
      if(m_set->IsEOF())
      {
        AfxMessageBox("当前视图没有记录!");
        return;
      }
    m_set-> MoveFirst();
    ReadDispalyRecord();
    UpdateData(false);
    }
}
```

### 8.4.7　浏览数据记录

**1.** 添加移动记录的消息响应函数

如图8.20所示，用“MFC ClassWizard”为“IDD_ODBCSQL_FORM”对话框中的4个Button控件（IDC_RECORD_FIRST、IDC_RECORD_NEXT、IDC_RECORD_PREV和IDC_RECORD_LAST）的BN_CLICKED事件添加消息响应函数：void OnRecordFirst()、void OnRecordNext()、void OnRecordPrev()和void OnRecordLast()。

**2.** 编写记录移动操作的程序代码

```
void COdbcsqlView::OnRecordFirst()
{
    m_set->MoveFirst();
        ReadDispalyRecord();
        UpdateData(false);
}

void COdbcsqlView::OnRecordLast()
{
    m_set->MoveLast();
```

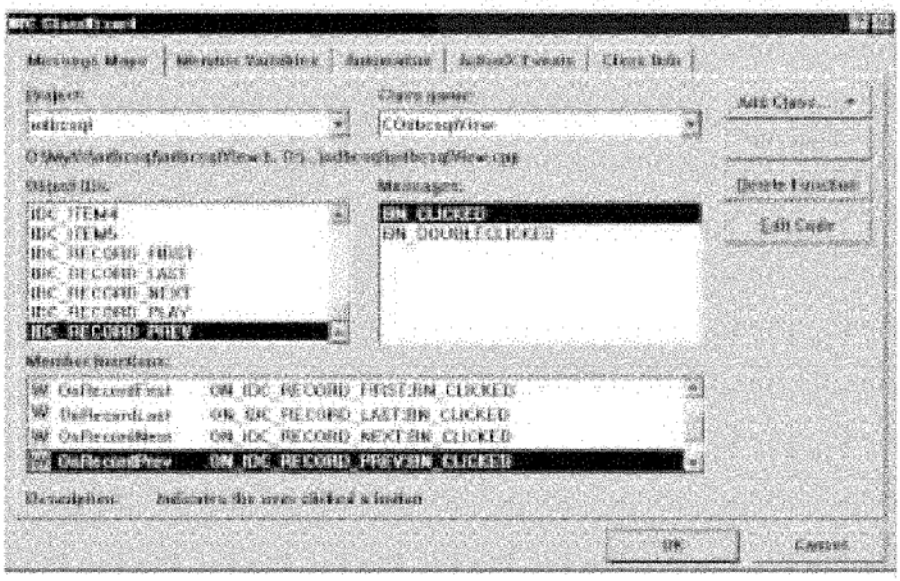



图 8.20 “MFC ClassWizard”对话框

```
    ReadDispalyRecord();
    UpdateData(false);
}

void COdbcsqlView::OnRecordNext()
{
    if(m_set->IsEOF())
    {   m_set->MoveLast();
        AfxMessageBox("当前视图没有记录!");
        return;
    }
    if(m_set->IsBOF()) m_set->MoveFirst();
    else               m_set->MoveNext();
    ReadDispalyRecord();
    UpdateData(false);
}

void COdbcsqlView::OnRecordPrev()
{
    m_set->MovePrev();
    if(m_set->IsEOF())m_set->MoveLast();
    if(m_set->IsBOF())m_set->MoveFirst();
    ReadDispalyRecord();
    UpdateData(false);
}
```

### 8.4.8　实现 SQL 查询

**1. 为查询条件编辑框“IDC _ EDIT _ SQL”引入变量**

用“MFC ClassWizard”为“IDD _ ODBCSQL _ FORM”对话框中的查询条件 Edit Box 控件“IDC _ EDIT _ SQL”引入关联变量：

```
CString m_sql;
```

**2. 为“执行查询”按钮编写程序代码**

用“MFC ClassWizard”为 Button 控件“IDC _ SQL _ OK”的 BN _ CLICKED 事件添加消息响应函数 OnSqlOk()，并编写如下代码，实现 SQL 条件查询：

```
void COdbcsqlView::OnSqlOk()
{  //① 用查询条件编辑框的值更新关联变量 m_sql
   UpdateData(true);
   CString m_Statement = m_sql;
   if( m_Statement == "") {MessageBox("没有 SQL 语句");return;}
   //② 如果记录集已打开，则关闭记录集
   if(m_set->IsOpen())
   {
      m_set->Close();
   }
   CString OldStr;
   OldStr = m_Query;//

   TRY
   {  //③ 执行一般的查询语句
      m_Statement.MakeUpper();
      if(m_Statement.Find("SELECT") == -1)
      {
         m_set->m_pDatabase->ExecuteSQL(m_Statement);
      }
      else
      {
         m_Query = m_Statement;
      }
      //④ 打开记录集
      if(! m_set->IsOpen())
      {
         m_set->Open(CRecordset::dynaset,m_Query);
      }
   }

   CATCH_ALL(e)
   {
```

```
            TCHAR Err[255];
            e->GetErrorMessage(Err,255);
            AfxMessageBox(Err);
            if(! m_set->IsOpen())
            {
               m_set->Open(CRecordset::dynaset,OldStr);
               m_Query = OldStr;
               m_Statement = OldStr;
            }
         }
         END_CATCH_ALL
         //⑤ 读出并显示数据库字段名
         ReadDisplayFields();
         //添加行:
         if(m_set->IsEOF())
         {
            AfxMessageBox("当前视图没有记录!");
            return;
         }
         m_set-> MoveFirst();
         ReadDispalyRecord();
         UpdateData(false);
      }
```

### 8.4.9 断开数据源

当结束程序时,必须关闭同数据库的连接,关闭记录集,删除记录集对象。所以,需添加事件 WM_DESTORY 的消息响应函数 OnDestroy(),在其中关闭同数据库的连接,关闭记录集,在析构函数中删除记录集对象。

**1. 为 WM_DESTORY 事件编写代码**

操作步骤是:

(1) 从"View"菜单中选择"ClassWizard"菜单项,弹出"MFC ClassWizard"对话框。

(2) 选择"Message Maps"标签,并进行如下设置:

```
Class name: COdbcsqlView
Object IDs: COdbcsqlView
Messages:   WM_DESTROY
```

(3) 单击"Add Function"按钮来增加新函数 OnDestroy()。

(4) 单击"Edit Code"按钮,定位于 OnDestroy()函数,编辑函数代码:

```
void COdbcsqlView::OnDestroy()
{
   CFormView::OnDestroy();
   // TODO: Add your message handler code here
   if(m_DSOK == FALSE)//如注册数据源没成功,则返回
```

```
    {
        return;
    }
    if(m_Db.IsOpen())//如建立了与数据库的连接,则关闭数据库
    {
        m_Db.Close();
    }
    if(m_set->IsOpen())//如打开记录集,则关闭记录集
    {
        m_set->Close();
    }
}
```

**2.** 删除记录集对象

```
COdbcsqlView::~COdbcsqlView()
{
    if(m_set != NULL)delete m_set;
}
```

### 8.4.10　构建并运行程序

查看结果的操作步骤如下：

(1) 选择 Build 菜单中的 Build odbcsql.exe(或 Build All)。

(2) 选择 Build 菜单中的 Execute odbcsql.exe

(3) 单击界面上的记录操作按钮，各 Edit Box 框内将显示对应的记录数据，如图 8.16 所示。

(4) 在编辑框中输入 SQL 语句，再点击“执行查询”按钮，则会关闭原打开的数据记录集，重新打开符合 SQL 查询条件的数据记录集，通过单击界面上的记录操作按钮可以验证查询结果。

## 8.5　简易媒体点播系统开发

程序 odbcsql.exe 与本章开始展现的图 8.1 相比，只差一个“播放”功能。为此，把第 7 章的多媒体播放器程序 MyPlayer 与程序 odbcsql 进行整合，在程序 odbcsql 中添加媒体播放功能，将待播放的媒体记录纳入数据库管理，即将 odbcsql 程序改造为简易媒体点播系统，如图 8.1 所示。

### 8.5.1　可视化设计

在图 8.18 主界面中添加标识符为 IDC_RECORD_PLAY 的“播 放”Button 按钮，如图 8.21 所示。

### 8.5.2　添加 CMCIClass 类

为了在程序 odbcsql 中，添加媒体播放功能，必须把 CMCIclass 类插入到 odbcsql 工程，操作方

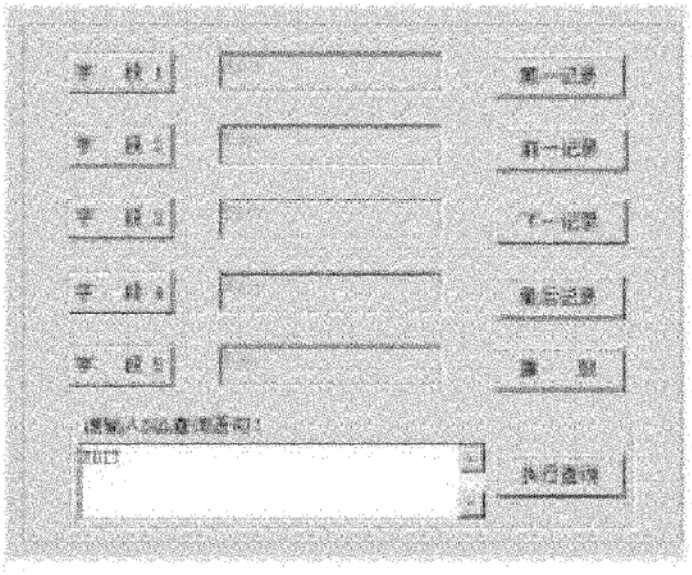


图 8.21 添加“播 放”按钮的主界面

法如下：

（1）将 CMCIclass 类的构成文件 MCIClass.h 和 MCIClass.cpp 拷贝到 odbcsql 工程目录下。

（2）从“Project”菜单中选择“Add To Project”菜单项，单击“File”弹出“Insert Files into Project”对话框。选中文件 MCIlass.cpp 和 MCIclass.h，如图 8.22 所示。单击“OK”按钮，就将 MCIlass.cpp 和 MCIclass.h 插入到了工程，也就完成了 CMCIclass 类的添加工作。

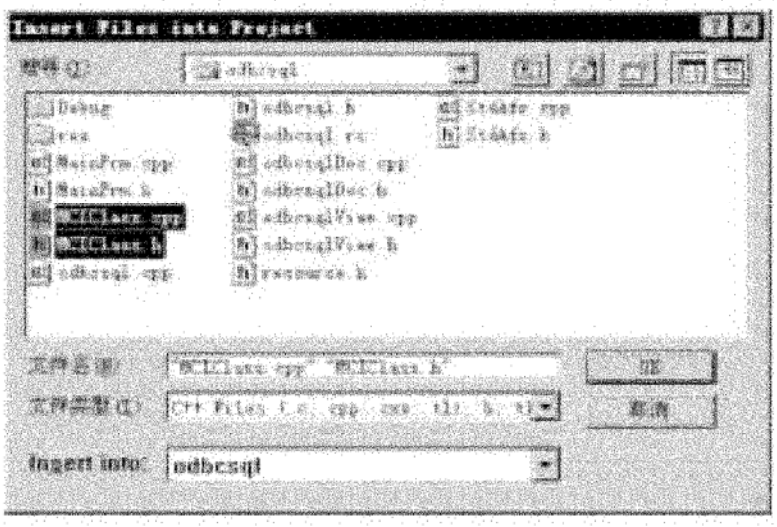



图 8.22 “Insert Files into Project”对话框

### 8.5.3 修改 COdbcsqlView 的基类

为了能在 COdbcsqlView 类中调用 CMCIClass 类的成员函数，简单的方法是把 CMCIClass 类作

为 COdbcsqlView 的基类，修改 COdbcsqlView 类的定义。为此，对文件 odbcsqlView.h 修改如下：

（1）在 odbcsqlView.h 文件中，添加如下两个头文件：

```
#include"MCIClass.h"
#include"mmsystem.h"
```

（2）将类 CMCIClass 作为类 COdbcsqlView 的基类，修改类的定义：

```
class COdbcsqlView : public CFormView,CMCIClass
{
protected: // create from serialization only
           COdbcsqlView();
           DECLARE_DYNCREATE(COdbcsqlView)
    ......
}
```

### 8.5.4　为“播放”按钮的 BN_CLICKED 事件编写代码

用 MFC ClassWizard 为“播放”按钮 IDC_RECORD_PLAY 添加消息响应函数，并编写如下代码：

```
void COdbcsqlView::OnRecordPlay()
{
    CString strFileName = m_position;   //获取当前记录 position 字段的媒体文件
  char szFileName[_MAX_PATH];
  GetShortPathName((LPCSTR)strFileName, szFileName, _MAX_PATH);
  char szExt[8];
  _splitpath((const char *)szFileName, NULL, NULL, NULL, szExt);
  // keep the open result
  MCIERROR mciError = 0;
  if(! stricmp(szExt,".wav"))
      mciError = Open(szFileName,"waveaudio"," ", m_hWnd);
  else if(! stricmp(szExt,".mid"))
      mciError = Open(szFileName,"sequencer"," ", m_hWnd);
  else if(! stricmp(szExt,".avi"))
      mciError = Open(szFileName,"avivideo","overlapped", m_hWnd);

  Play();   //调用 CMCIClass 类的方法，播放媒体
}
```

### 8.5.5　修改工程设置、构建并运行程序

**1.** 修改工程设置

从“Project”菜单中选择“Setting”菜单项，弹出“Project Settings”对话框。选中“Link”标签，在“Object/library modules”编辑框中输入 winmm.lib，如图 8.23 所示。单击“OK”按钮，就为工程 odbcsql 添加了多媒体库 winmm.lib 的连接设置。

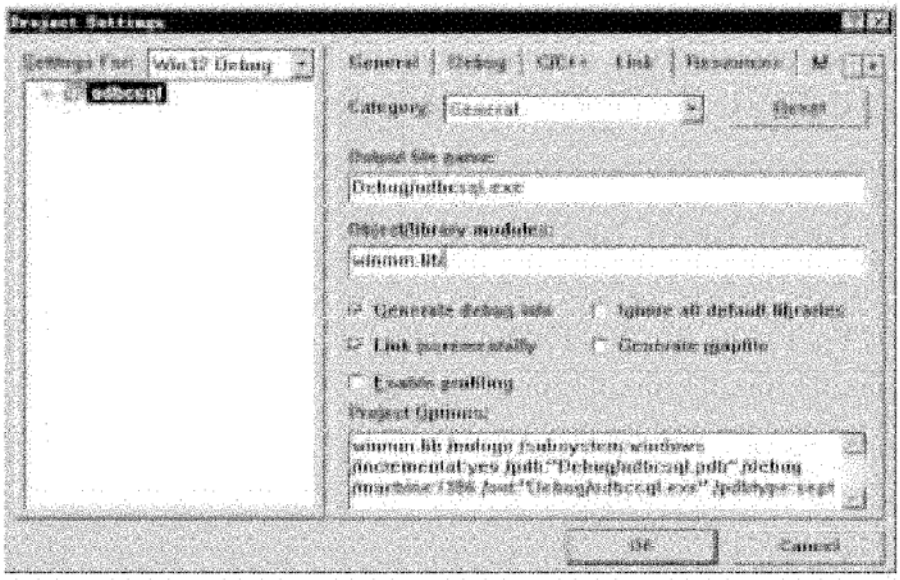



图 8.23 “Project Settings”对话框

**2.** 编译、连接运行

（1）选择 Build 菜单中的“Build odbcsql.exe(或 Build All)”。

（2）选择 Build 菜单中的“Execute odbcsql.exe”。

（3）单击界面上的记录操作按钮，各 Edit Box 框内将显示对应的记录数据，如图 8.1 所示。

（4）在编辑框中输入 SQL 语句，在点击“执行查询”按钮，则会关闭原打开的数据记录集，重新打开符合 SQL 查询条件的数据记录集，通过单击界面上的记录操作按钮可以验证查询结果。

（5）单击界面上的“播放”按钮，程序就会播放当前记录媒体。

## 习 题 八

1. 请修改 odbcsql.exe 程序，为系统操作员提供记录编辑功能(要求密码验证)。

2. 进一步修改 odbcsql.exe 程序，添加进程条、暂停和恢复等功能。

# 第9章

# 网络编程

本章导读

要实现网络编程，有多种方法：通过 WinSock、使用命名管道和邮箱、使用 RPC（RemoteProceedure Call）等均可以实现在网络上进行通信。

本章将介绍利用 MFC 提供的 WinSock 类进行编程，即使用 MFC 提供的 CAsyncSocket 和 CSocket 类实现网络编程。通过编写一个很实用、比较复杂的网络应用程序：基于客户/服务器模式的公众聊天室，使读者：

- 熟悉 CSocket 程序设计的基础
- 掌握基于 CSocket 类的网络编程技术

---

## 9.1 聊天室程序

### 9.1.1 聊天室应用程序功能介绍

**1.** 服务器应用程序

服务器应用程序启动时，需要指定提供聊天服务的端口号是多少（采用默认值 800，也可以是其他值，但必须与客户端的应用程序保持一致）。

图 9.1 演示了服务器端应用程序的界面。它含有一个运行情况，也就是监控该聊天室所有聊天客户发送的文字信息。服务器的任务就是负责接收来自客户端的聊天信息，并将这些信息转发到另外一个或多个聊天客户。

**2.** 客户端应用程序

客户端应用程序开始运行时，先进行聊天室设置：输入聊天室服务器的 IP 地址和聊天频道（默认值 800），同时还要输入自己的登入名。然后单击“连接”按钮与聊天服务器建立连接。

图 9.2 演示了聊天室某一客户端应用程序的界面。界面主窗口有 3 个编辑框：

（1）编辑发送信息　编辑聊天文字。

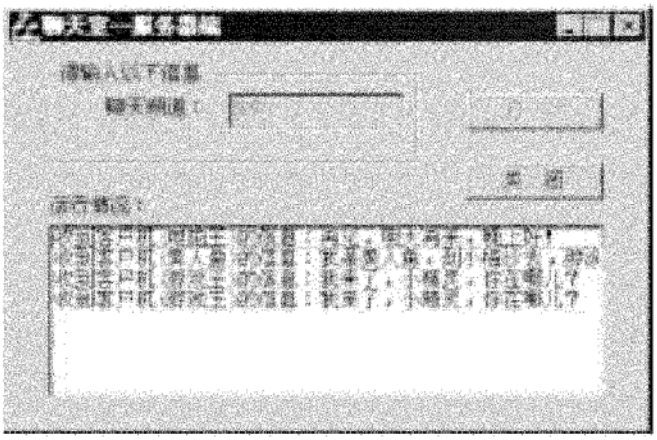


图 9.1 聊天室服务器端

(2) 已发送的信息 显示已给网友发送的文字信息。

(3) 聊天室信息 聊天室中,其他聊天客户发送的文字信息。

特别说明:

当读者在自己的计算机上运行时,需要指定的服务器地址要根据自己的实际情况进行设置。如果没有两台电脑,也可以把这个地址设置成自己计算机的 IP 地址。

为了避免输入计算机的 IP 地址带来的麻烦,可以用计算机名来代替 IP 地址(参见 9.2.1)。

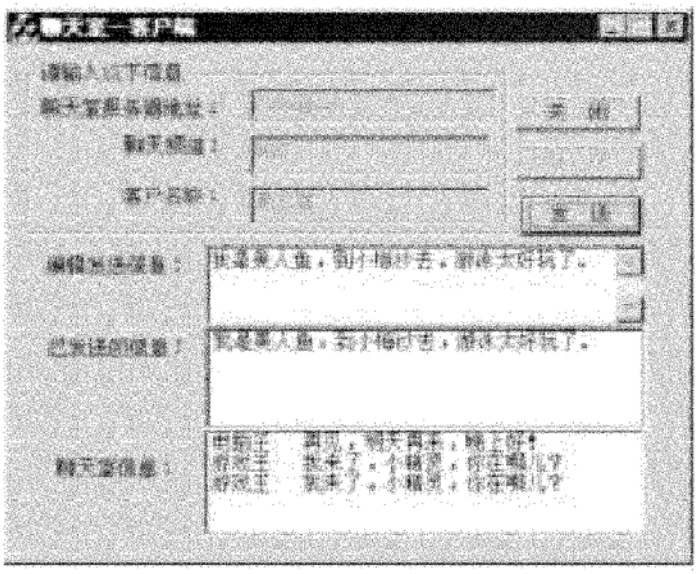


图 9.2 聊天室某客户端

### 9.1.2 程序开发步骤

将目标程序按知识点和功能进行分解,按以下几步完成:

(1) 介绍 CStocket 程序设计基础。

（2）使用 CSocket 类，开发面向有连接的流式套接口客户端网络通信程序，即聊天室客户端应用程序。

（3）使用 CSocket 类，开发面向有连接的流式套接口服务器端网络通信程序，即聊天室服务器端应用程序。

## 9.2　CSocket 程序设计基础

在 VC++ 中，利用 WinSock 编程主要有两种途径：一种是通过 WinSock API 进行编程，另一种是利用 MFC 提供的 WinSock 类编程，本章主要介绍后者。为此，先对有关概念作扼要的介绍。

### 9.2.1　计算机名、IP 地址和端口

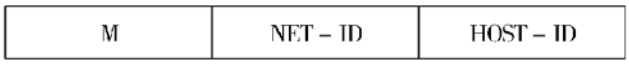
**1. IP 地址**

IP 地址是一个 32 位的数字，用于惟一地标识位于网络中的计算机。IP 地址由两部分组成：网络标识和主机标识。利用 IP 地址寻找计算机时，首先根据 IP 地址的网络标识找到计算机所在的网络，然后再根据主机标识找到指定的计算机。IP 地址的格式有两类：二进制和十进制格式。其中十进制是由二进制翻译过去的，并通常将 32 位二进制的 IP 地址以 8 为单位进行分隔，换算成十进制，每个十进制数之间用“.”隔开，例如

二进制的 IP 地址：10100010，1110010，1011111，10000001

转化为十进制的 IP 地址是：162.114.95.129

由于各个计算机所处的网络差异很大，有的网络中计算机数目很多，而有的则很少，因此，为了便于对 IP 地址进行管理，将 IP 地址分为 5 类，即 A、B、C、D、E 五类 IP 地址。其一般格式如下：




其中，M 为类别号，NET－ID 为网络号，HOST－ID 为主机号。这 3 个参数所占位数由 IP 地址的类型所决定。目前所使用的 IP 地址大多数为 C 类地址。五类地址的构成如下图 9.3 所示。

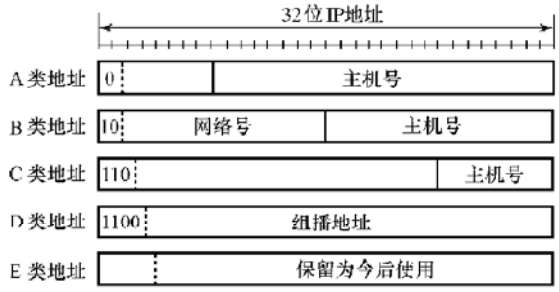



图 9.3　五类 IP 地址的构成

**2.** 计算机名

为了避免输入计算机的IP地址带来的麻烦，可以通过计算机名来代替IP地址。对于Windows 2000，利用 \ winnt \ Sysmem32 \ drivers \ etc 下的 hosts 文件可以实现这一点。只要将 hosts.asm 拷贝成 hosts 即可实现计算机名同IP地址的关联。典型的 hosts 文件的内容如下：

```
127.0.0.1     localhost      # localhost 为计算机名
```

**3.** 端口

在利用计算机网络进行通信时，不仅需要利用IP地址找到指定的计算机，还应该通过某种方式来标识进行通信的进程。TCP/UDP协议通过端口(Port)来标识进行通信的进程，TCP/UDP协议为每个端口分配一个端口号，当进行通信的进程通过系统调用，同一个或多个端口建立联系后，就可已通过这些端口进行数据传输了。

TCP/IP端口号有16位，因此可以有$2^{16}$个端口。在这$2^{16}$个端口中，有一些是保留端口，由权威机构分配，用于特殊目的。例如，端口80被用作超文本协议和WWW服务。除了众所周知的保留端口之外的端口为自由端口，以本地方式进行分配。

### 9.2.2 WinSock和MFC

通信的基础是套接口(Socket)，一个套接口是通信的一端。在这一端上用户可以找到与其对应的名字。一个正在被使用的套接口都有它的类型和与相关的进程。

用户目前可以使用两种套接口，即流套接口和数据报套接口。流套接口提供了双向的、有序的、无重复并且无记录边界的数据流服务。数据报套接口支持双向的数据流，但并不保证是可靠、有序和无重复的。也就是说，一个数据报套接口接收信息的进程有可能发现信息重复了，或者和发出时的顺序不同。数据报套接口的一个重要特点是它保留了记录边界。

数据报套接口可以用来向许多系统支持的网络发送广播数据包。要实现这种功能，网络本身必须支持广播功能，因为系统软件并不提供对广播功能的任何模拟。

在MFC中，提供了两个类用于WinSock编程：CAsyncSocket类和CSocket类。其中，CAsyncSocket类对Windows Sockets API进行了封装，提供基于异步通信的套接口封装功能，该类适用于需要充分利用WinSock API的灵活性，并使用回调函数来响应网络事件的情况；而CSocket类是CAsyncSocket类的派生类，该类对WinSock类进行了更多的封装，提供更加高层次的功能，并且利用CArchive类进行数据传输，从而使利用该类进行数据传输的过程与MFC的串行化(Serialize)一致。此外，该类通过对Windows消息的处理来实现阻塞，从而实现基于CAchive的同步数据传输。

在编写网络程序时，所创建的新类必须声明自己是CAsyncSocket或CSocket的子类，其主要原因是想捕捉激活的事件，如消息收到、连接完成等。有一系列函数可以调用，以响应每一个事件。这些函数使用同样的定义，只是函数名有些差别，所有函数声明成CAsyncSocket类的被保护成员，很可能在用户的派生类中也应该声明成被保护的函数。这些函数有一个整型参数，是一个错误码，应加以核对确保没有错误。表9.1列出CAsyncSocket事件函数及其表示的事件。

表 9.1　CAsyncSocket 可重载事件响应函数

| 函　数 | 事件描述 |
| --- | --- |
| OnAccept | 监听方调用此函数表示对方程序的连接请求正等待被接受 |
| OnClose | 表示连接的另一端的应用程序已经关闭它的 Socket，或者连接已经丢失，接着，收到此通知的 Socket 应该关闭 |
| OnConnect | 表示与对方程序的连接已经完成，程序现在可以通过 Socket 发送或接收消息 |
| OnOutofBanData | 表示收到外带数据，外带数据在一个逻辑上独立的通道上发送，用于经济数据的发送，不是常规数据的一部分，Send 和 Receive 方法都用第 3 个参数，用于外带数据的发送和接收 |
| OnReceive | 表示通过 Socket 连接的数据已经收到，可调用 OnReceive 函数接收 |
| OnSend | 表示通过 Socket 已经准备好发送数据，连接完成即可调用此函数。通常，程序传送给 Send 函数的数据太多，超过了单个数据包中可以发送的数量，这时也可以调用此函数。这表示，已发送所有数据，程序可以发送下一个缓冲数据 |

除了这些可以重载的事件响应函数外，CSocket 还提供了一个函数 OnMessagePending( )，在应用程序事件消息队列中有消息等待事件调用此函数。在找到特定的 Windows 消息后，用 CSocket 类响应之。

### 9.2.3　WinSock 的工作原理

WinSock 的工作方式为服务器—客户模式，而根据连接的方式不同，又分为面向连接和无连接两种通信方式。

**1.** 利用 **WinSock** 进行有连接的通信

面向连接的通信方式使用的是 TCP(传输控制协议)协议，通过 TCP 协议可以与指定 IP 地址的主机建立连接，同时利用建立的连接可以双向交换数据。这种通信方式采用流的方式进行数据传输，要求客户必须指定服务器的地址，其优点是保证通过网络进行数据传输的可靠性，它会检查数据是否到达，对没有到达的数据会进行重传，还会通过检验校验和的方式确定传来的数据是否正确，典型的面向连接的程序的工作方式如图 9.4 所示。对于有连接的通信，不论是数据的发送还是接收，其顺序都是有保证的。

从图 9.4 可以看出，在有客户连接请求之前，服务器线程或进程将会阻塞，直到有客户程序成功地连接到了服务器上后才继续往下运行。此外，在进行读/写操作时，当数据读/写因为种种原因尚未完成时，服务器和客户线程或进程也将发生阻塞，在单线程的应用程序中，这意味着整个应用程序都无法继续运行。这可以通过选择多线程的方式来解决上述问题，即：对于服务器而言，每当需要进入等待时，都创建一个辅助线程，这样对应于每一个客户，都分配一个线程；而对于客户应用程序，则在辅助线程中完成所有读/写操作。对服务器而言，采用这种解决方式的服务器通常被叫做基于并发的、面向连接的服务器。

是否还有别的解决阻塞的方案呢？答案是肯定的，那就是利用产生和响应网络事件的方式来解决阻塞问题，这种解决方案的意思是，服务器在等待客户请求时并不进入阻塞状态，而是直接返回，进入侦听网络事件的循环中。当发现了客户的连接请求时，服务器将收到一个与客户连

接请求相关的操作，然后又返回。当需要读客户数据时，同样也并不陷入阻塞状态，而是直接返回到消息循环中，当客户应用程序有数据发送过来时，服务器接收到一个与读入数据相关的网络事件，于是服务器调用相关的消息响应函数，进行数据读入操作。在客户方，连接到服务器和数据的读取也遵循同样的模式，这样，无论何时，应用程序都不会陷入阻塞。

由于采用这种算法时，服务器和客户的操作并没有保持同步，因此，在 WinSock 中，将使用网络事件的应用程序叫做基于异步通信的应用程序。MFC 中的 CAsyncSocket 类和 CSocket 类就支持异步通信模式。

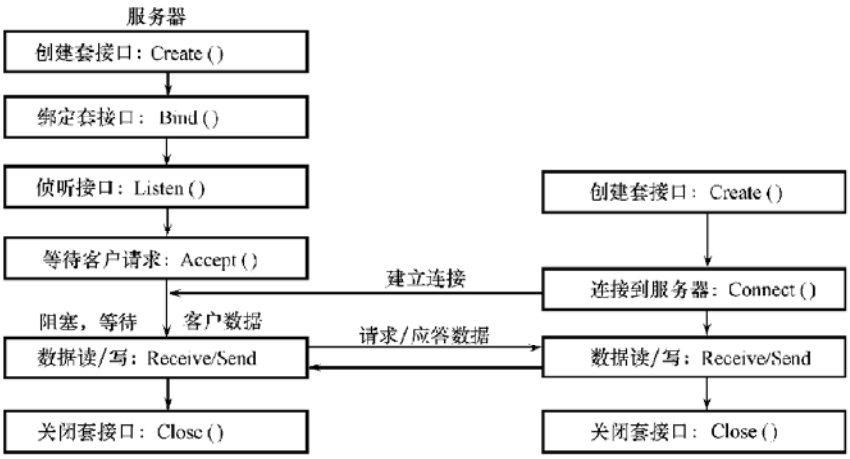



图 9.4 WinSock 进行有连接的通信模型

**2.** 利用 **WinSock** 进行无连接的通信

无连接的通信方式则使用的是 UDP(用户数据报协议)协议，通过 UDP 协议可以往指定 IP 地址的主机发送数据，同时也可以从指定 IP 地址的主机接收数据，发送和接收方处于相同的地位，没有主次之分。

利用 CSocket 操作无连接的数据发送很简单，首先生成一个本地套接口(需要指明 SOCK_DGRAM 标记)，然后利用如下语句：

```
int CAsyncSocket::SendTo(cont void * lpBuf, int nBufLen, UINT nHostPort,
                         LPCTSTR lpszHostAddress = NULL, int nFlag = 0);
```

发送数据。用如下语句：

```
int CAsyncSocket::ReceiveFrom(void * lpBuf, int nBufLen,
          CString & rSocketAddress, UINT& rSocketPort, int nFlags = 0);
```

接收数据。函数调用顺序如图 9.5 所示。

利用 UDP 协议发送和接收都可以是双向的，就是说任何一个主机都可以发送和接收数据。但是 UDP 协议是无连接的，所以发送的数据不一定能被接收，此外接收的顺序也有可能与发送的顺序不一致。

例如，发送方在端口 800 上向接收方端口 900 发送数据的部分子函数如下：

```
//发送方代码
```

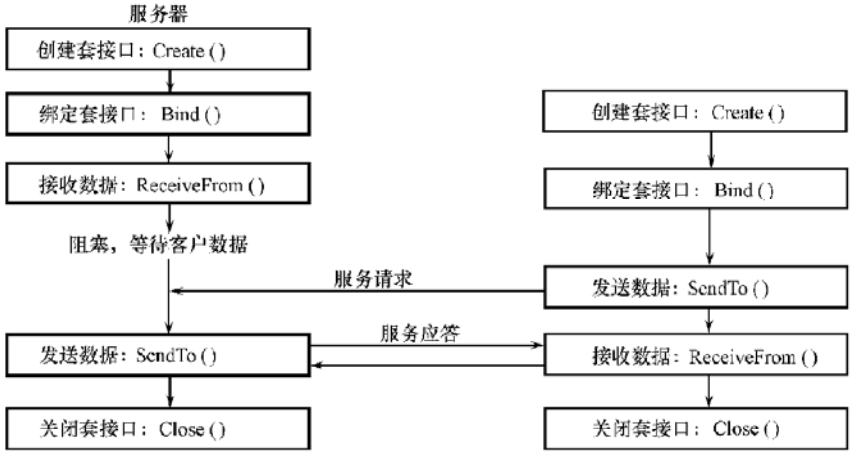



图 9.5　WinSock 讲行无连接的通信模型

```
// m_port:800,m_ip = _T("127.0.0.1");void CMyClientDlg::OnLink()
{
   // TODO: Add extra initialization here
   UpdateData(true);
   m_socketRecv.Create(m_port,SOCK_DGRAM,m_ip);
   m_socketRecv.Bind(m_port,m_ip);
   SetTimer(2,3000,NULL);
   ......
}
void CMyClientDlg::OnTimer(UINT nIDEvent)
{
   char szRecv[20];

   int iRecv = m_socketRecv.ReceiveFrom(szRecv,10,m_ip,m_port,0);
   szRecv[iRecv] = '\0';
   m_edit1 = szRecv;
   UpdateData(false);
   CDialog::OnTimer(nIDEvent);
}

void CMyserverDlg::OnLink()
{
   UpdateData(true);
   m_socketSend.Create(m_port,SOCK_DGRAM,NULL);
   m_socketSend.Bind(m_port,m_ip);
   SetTimer(1,3000,NULL);
   ......
}

void CMyserverDlg::OnTimer(UINT nIDEvent)
{
   static iIndex = 0;
```

```
    char szSend[20] = "1234567890";
    sprintf(szSend, "%d", iIndex++);
    m_transmit_info = szSend;
    UpdateData(false);
    int iSend = m_socketSend.SendTo(szSend,10,m_port,m_ip,0);
    CDialog::OnTimer(nIDEvent);
}
```

## 9.3 基于 CSocket 的网络编程

面向有连接的流式套接口提供了可靠的、有序的且不重复的连接，在进行连接时，服务器端通常在一个众所周知的端口侦听，当客户端发出连接请求时，服务器被唤醒，然后为客户应用程序提供服务或进行数据传输，这个过程同我们平常打电话非常相似。

使用 MFC 编制基于流式套接口的应用程序的最简单的方法是使用 CSocket 类编制基于异步通信模式的应用程序。

**1.** 服务器端

使用 CSocket 类编制网络端应用程序的步骤为：

（1）从 CSocket 类派生一个新类。

（2）创建一个 CSocket 派生类对象。

（3）调用 CSocket::Create()函数创建一个套接口，并指定一个端口。

（4）调用 CSocket::Listen()函数侦听端口。

（5）在 CSocket 派生类中添加虚函数 OnAccept()。当有客户应用程序发出连接请求时，该函数用于响应客户应用程序的连接请求。通过调用 CSocket::OnAccept()函数接受客户的连接请求，然后，在用于读写的 CSocket 派生类的 OnReceive()函数中进行读写操作。具体而言，服务器端要同时处理多个客户端的请求，与多个客户端实现并行通信，应该在 OnAccept()函数中为每一个连接进来的客户创建一个新的 CSocket 类对象，以便通信时不致于发生混乱。而且这样，从客户端看，好象服务器只有自己一个客户端连接在上面，独占了服务器。

（6）通信结束时，调用读写套接口的 Close()函数关闭为各个客户分配的读写套接口。

**2.** 客户端

客户端的异步通信模式实现过程如下：

（1）创建一个 CSocket 类的派生类对象，用于连接和读写。

（2）调用 CSocket::Create()函数创建一个套接口。

（3）调用 CSocket::Connect()函数连接服务器的指定端口。

（4）调用 CSocket::Send()函数，发送数据。

（5）在 CSocket::OnReceive()函数中进行读写操作。

（6）在结束通信时，调用 CSocket::Close()函数关闭套接口。

下面将开发基于流式套接口的网络应用程序：基于客户/服务器模式的公众聊天室。也就是

要开发具有以下功能网络应用程序：

• 客户端应用程序将向服务器发送字符串信息，服务器收到后，立即显示收到的内容，并将收到的内容发送给所有客户端；客户端收到服务器的信息后，将其收到的内容在主窗口显示。

• 服务器可与多个客户端进行通信。

## 9.4　聊天室客户端应用程序

### 9.4.1　创建工程 MyWc

创建 MyWc 工程的步骤如下：

（1）启动 Visual C++6.0。从“File”菜单中选择“New”。此时，Visual C++将显示一个“New”对话框。

（2）在“New”对话框中选择“Project”标签。然后选择“MFC AppWizard[exe]”类型，告诉 Visual C++即将创建一个 EXE 程序。

（3）在“New”对话框的“Project name”文本编辑框中输入“MyWc”，单击位于“Location”框右边的小按钮，再从下拉的对话框中选择“D:\MYVC”目录，使新创建的工程文件放置在“D:\MYVC”目录之下。

（4）单击“New”对话框的“OK”按钮。此时 Visual C++将显示“MFC AppWizard - Step 1”对话框。

（5）在“MFC AppWizard - Step 1”对话框中，选择“Dialog based”，创建一个基于对话框的应用程序，然后单击“Next”按钮。

（6）在“MFC AppWizard - Step 2 of 4”对话框中，选择“Windows Sockets”支持。其他接受默认设置，如图 9.6 所示。

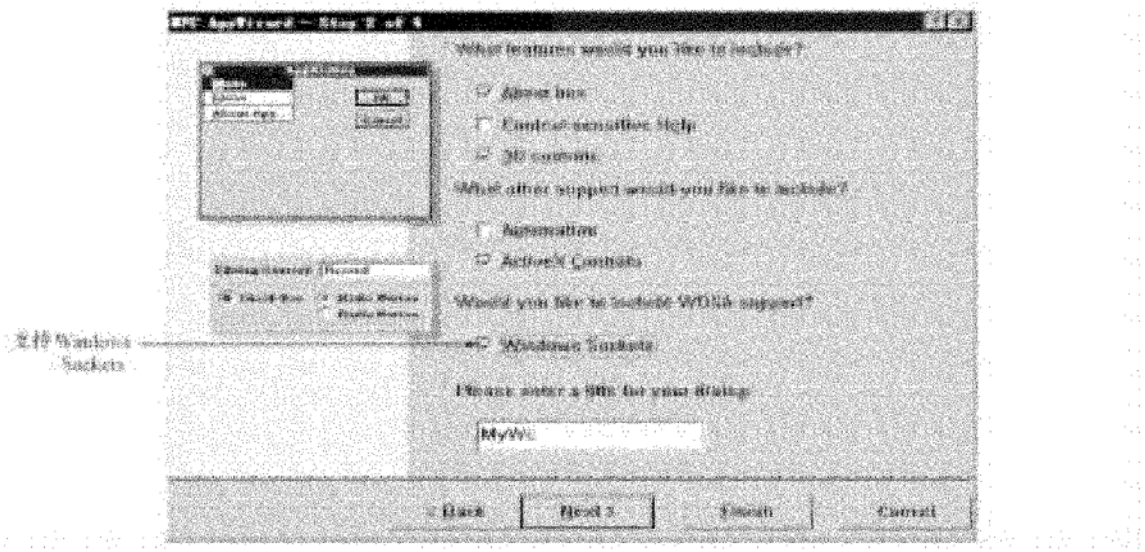


图 9.6　选择支持“Windows Sockets”对话框

（7）然后单击“Finish”按钮，此时 Visual C++将显示“NewProject Information”窗口。

（8）单击“OK”，于是，Visual C++就会创建 MyWc 工程以及相关的所有文件。

### 9.4.2 可视化设计

按照图 9.2 用户界面，设计主窗口界面的步骤如下：

（1）在“Project Workspace”窗口中，单击“ResourceView”标签，把 MyWc resources 扩展开，然后再扩展 Dialog，最后，双击“IDD _ MYWC _ DIALOG”项，Visual C++显示出处于设计状态的“IDD _ MYWC _ DIALOG”对话框。

（2）在“IDD _ MYWC _ DIALOG”对话框中，根据表 9.2 中的控件属性，编辑对话框资源，设计完毕后对话框如图 9.7 所示。

表 9.2 控件属性表

| 对 象 | 属 性 | 设 置 |
|---|---|---|
| Button | ID<br>Caption | IDC _ CLOSE<br>关 闭 |
| Button | ID<br>Caption | IDC _ CONNECT<br>连 接 |
| Button | ID<br>Caption | IDC _ SEND<br>发 送 |
| List Box | ID<br>Selection<br>Vertical Scroll | IDC _ RECEIVED _ INFO<br>Muliline（Styles 标签）<br>Checked（Styles 标签） |
| List Box | ID<br>Selection<br>Vertical Scroll | IDC _ SENT _ INFO<br>Muliline（Styles 标签）<br>Checked（Styles 标签） |
| Edit Box | ID | IDC _ SERVER _ NAME |
| Edit Box | ID | IDC _ SERVER _ PORT |
| Edit Box | ID | IDC _ CLIENT _ NAME |
| Edit Box | ID<br>Multiline<br>Auto Hscroll<br>Want return | IDC _ SEND _ INFO<br>Checked（Styles 标签）<br>Checked（Styles 标签）<br>Checked（Styles 标签） |

### 9.4.3 创建一个新类 CWCSock

在 MyWc 工程中，创建一个 CSocket 类的派生类 CWCSock，用于同服务器通信。操作步骤如下：

（1）选择“Insert”菜单的“New Class”菜单项，弹出“New Class”对话框。

（2）在“New Class”对话框中的“Base class”下拉式列表框中选取“CSocket”作为 CWCSocket 的基类。如图 9.8 所示。

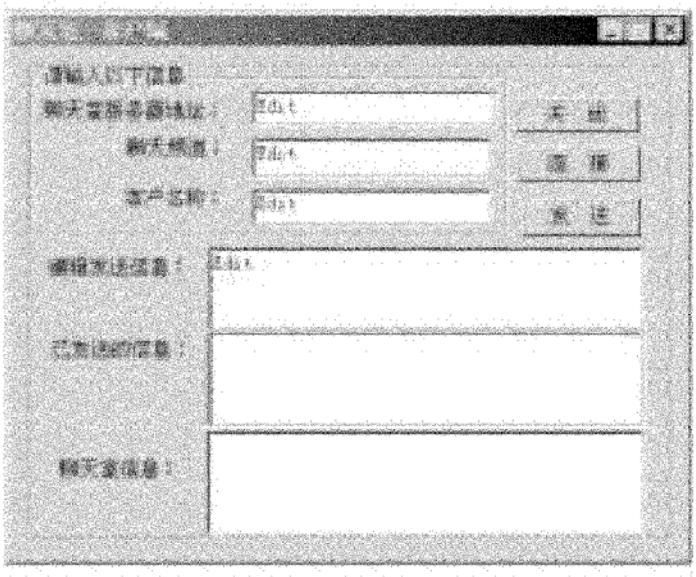


图 9.7　设计完后的“IDD _ MYWC _ DIALOG”对话框

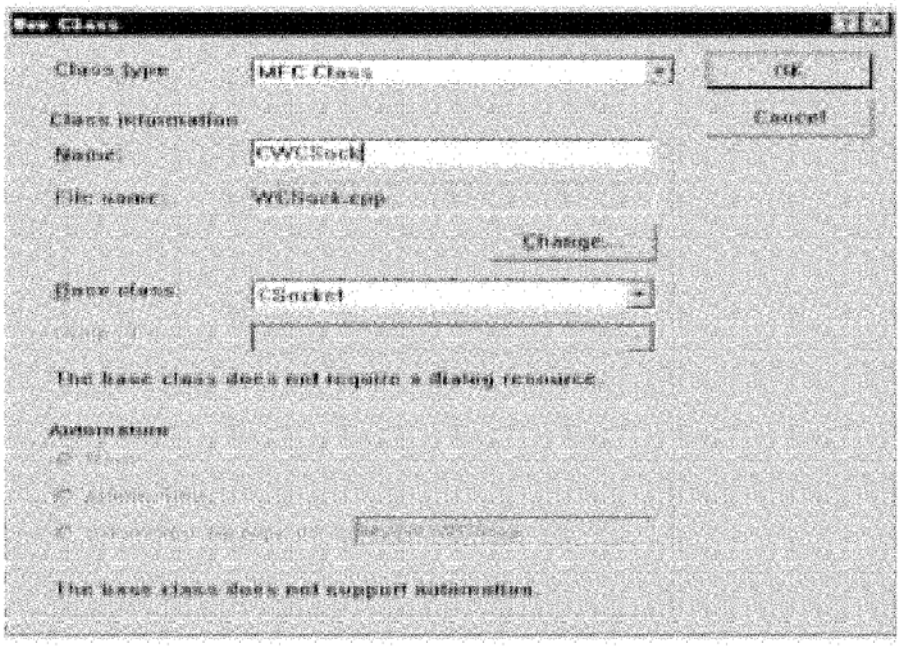



图 9.8　创建新类 CWCSock 的对话框

（3）单击“OK”按钮，VC ++ 就为 MyWc 工程创建了一个新类 CWCSocket，相应地为工程创建了两个文件 WCSocket.h 和 WCSocket.cpp。

### 9.4.4　修改 CWCSocket 类

按照下面的方法修改 CWCSocket 类，以便与服务器通信。

（1）在 WCSock.h 中定义一个与服务器端相同的结构，以便通过该结构的对象存储数据。

```
typedef struct
{
   char m_UserName[255];          //客户名字
   char m_dbData[255];            //要发送的数据或信息
   BOOL m_bOnline;                //客户是否在线
}_DATA;
```

（2）在 CWCSock 类中，添加_DATA 类型的成员变量 m_Dat，用于存储数据。

```
class CWCSock : public CSocket
{
// Attributes
public:

// Operations
public:
      CWCSock();
      virtual ~CWCSock();

// Overrides
public:
      _DATA m_Dat;
      // ClassWizard generated virtual function overrides
      //{{AFX_VIRTUAL(CWCSock)
      public:
      //}}AFX_VIRTUAL

      // Generated message map functions
      //{{AFX_MSG(CWCSock)
           // NOTE - the ClassWizard will add and remove member functions here.
      //}}AFX_MSG
// Implementation
protected:
};
```

（3）在 WCSock.h 中，定义自定义消息，以使 CWCSock 能够同应用程序主窗口通信。

```
#define RE_RECEIVED WM_USER+1
```

（4）如图 9.9 所示，用“MFC ClassWizard”在 CWCSock 类中添加虚函数 OnReceive()，在该函数中读取服务器传来的数据，并通知应用程序主窗口进行相应的处理：

```
void CWCSock::OnReceive(int nErrorCode)
{
   // TODO: Add your specialized code here and/or call the base class
   if(Receive(&m_Dat,sizeof(m_Dat))==sizeof(m_Dat))
   ::PostMessage(::AfxGetApp()->m_pMainWnd->m_hWnd,
                    RE_RECEIVED,(WPARAM)&m_Dat,0);
```

```
    CSocket::OnReceive(nErrorCode);
}
```

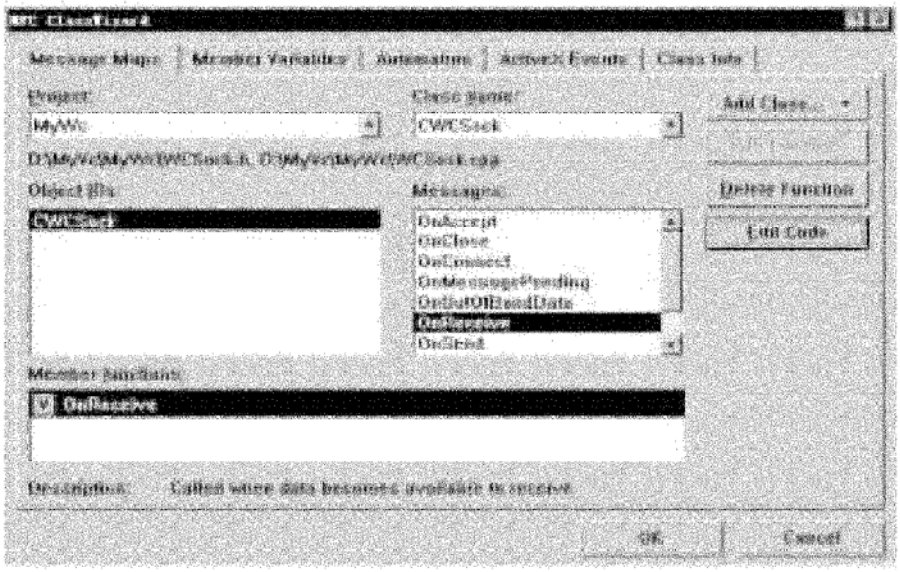


图9.9　添加虚函数 OnReceive()

### 9.4.5　为编辑控件引入变量

在程序运行时，为了将编辑框中的数据发送，或将从服务器接收到的数据显示在对应的编辑框中，必须为它们引入变量。

用“MFC ClassWizard”，按表9.3所列属性为各编辑控件引入关联变量，如图9.10所示。

表9.3　各编辑控件的关联变量属性表

| 控件 | ID标识符 | 变量类型 | 变量名 |
|---|---|---|---|
| Edit Box | IDC_CLIENT_NAME | CString | m_client_name |
| List Box | IDC_RECEIVED_INFO | CListBox | m_ctlRecvd |
| Edit Box | IDC_SERVER_NAME | CString | m_server_name |
| Edit Box | IDC_SERVER_PORT | int | m_nPort |
| Edit Box | IDC_SEND_INFO | CString | m_send_info |
| List Box | IDC_SENT_INFO | CListBox | m_ctlSent |

接着做下列工作：

（1）为CMyWcDlg类添加相应的成员变量

```
class CMyWcDlg : public CDialog
{
// Construction
public:
```

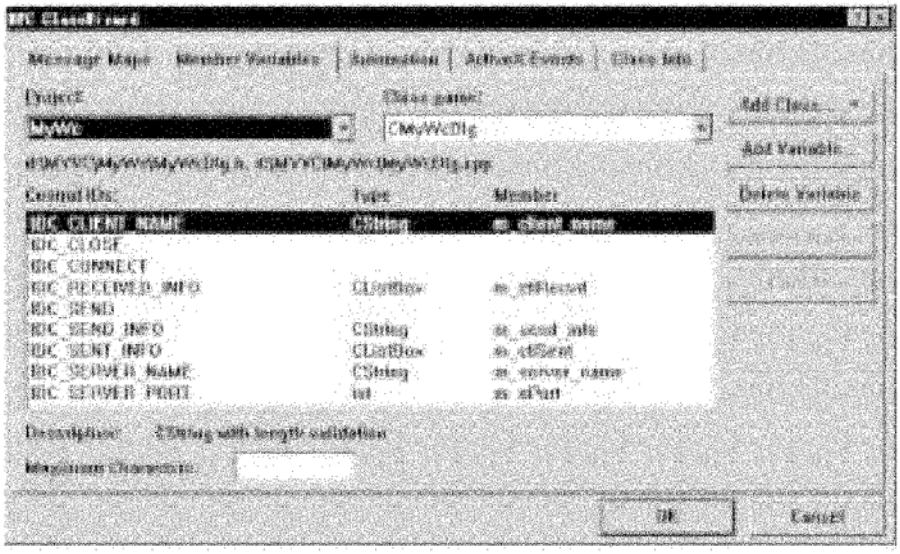


图 9.10 各编辑控件附上关联变量

```
    CMyWcDlg(CWnd * pParent = NULL); // standard constructor
    CWCSock sockClient;
    _DATA m_Dat;

    LRESULT OnReceive(WPARAM wparam,LPARAM lParam);
// Dialog Data
    //{{AFX_DATA(CMyWcDlg)
    enum { IDD = IDD_MYWC_DIALOG };
    CListBox   m_ctlSent;
    CListBox   m_ctlRecvd;
    int        m_nPort;
    CString    m_server_name;
    CString    m_client_name;
    CString    m_send_info;
    //}}AFX_DATA
    ......
    DECLARE_MESSAGE_MAP()
};
```

（2）在构造函数中进行初始化

```
CMyWcDlg::CMyWcDlg(CWnd * pParent /* = NULL */)
   : CDialog(CMyWcDlg::IDD, pParent)
{
   memset(&m_Dat,0,sizeof(m_Dat));
   //{{AFX_DATA_INIT(CMyWcDlg)
   m_nPort = 800;
   m_server_name = _T("localhost");
   m_client_name = _T("localhost");
   m_send_info = _T("");
```

```
    //}}AFX_DATA_INIT
    // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
    m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

（3）添加数据交换代码

```
void CMyWcDlg::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CMyWcDlg)
    DDX_Control(pDX, IDC_SENT_INFO, m_ctlSent);
    DDX_Control(pDX, IDC_RECEIVED_INFO, m_ctlRecvd);
    DDX_Text(pDX, IDC_SERVER_PORT, m_nPort);
    DDX_Text(pDX, IDC_SERVER_NAME, m_server_name);
    DDX_Text(pDX, IDC_CLIENT_NAME, m_client_name);
    DDX_Text(pDX, IDC_SEND_INFO, m_send_info);
    //}}AFX_DATA_MAP
}
```

### 9.4.6　编写程序代码

**1.** 添加消息响应函数

用“MFC ClassWizard”为“IDD_MYWC_DIALOG”对话框中的 Button 控件（IDC_SEND、IDC_CONNECT 和 IDC_CLOSE）的 BN_CLICKED 事件添加消息响应函数：

```
void OnSend()、void OnConnect()、void OnClose()
```

**2.** 编写程序代码

```
void CMyWcDlg::OnConnect()
{
    // TODO: Add your control notification handler code here
    //更新关联变量，如服务器名、端口号
    UpdateData(true);
    //创建套接口
    if(! sockClient.Create())
    {
        AfxMessageBox("Create WC socket failed");
        PostQuitMessage(0);
    }
    //与服务器连接
    if(! sockClient.Connect(m_server_name,m_nPort))
    {
        MessageBox("连接失败!");
        return;
    }
```

```
    m_Dat.m_bOnline = true;
}

void CMyWcDlg::OnSend()
{  //更新关联变量,如要发送的信息
   UpdateData(true);

   m_Dat.m_bOnline = true;
   memset(m_Dat.m_cbData,0,255);
   memcpy(m_Dat.m_cbData,m_send_info,m_send_info.GetLength());

   strcpy(m_Dat.m_strName,m_client_name);
   //发送的信息
   int iSent = sockClient.Send(&m_Dat,sizeof(m_Dat));
   //如果发送成功,将其发送的信息添加并显示在列表编辑框
   if(iSent != SOCKET_ERROR)
   {  m_ctlSent.AddString(m_send_info);
      UpdateData(false); // 显示
   }
}

void CMyWcDlg::OnClose()
{
   // TODO: Add your control notification handler code here
   m_Dat.m_bOnline = false;

   strcpy(m_Dat.m_dbData,"对不起,我要走了,再见!");
   sockClient.Send(&m_Dat,sizeof(m_Dat));
   //断开与服务器的连接
   sockClient.Close();
}
```

### 9.4.7 建立 CMyWcDlg 类与 CWCSock 类的关联

在 CMyWcDlg 类中添加 CWCSock 类的对象,使它们相互关联。方法是在 MyWcDlg.h 文件中,在 CMyWcDlg 类定义前面添加如下语句:

```
#include"WCSock.h"
```

在 CMyWcDlg 类定义中添加如下语句:

```
CWCSock sockClient;
```

### 9.4.8 处理自定义消息

为响应 CWCSock 类对象发送过来的自定义消息,添加如下的消息映射宏:

```
BEGIN_MESSAGE_MAP(CMyWcDlg, CDialog)
```

```
    //{{AFX_MSG_MAP(CMyWcDlg)
    ON_WM_SYSCOMMAND()
    ON_WM_PAINT()
    ON_WM_QUERYDRAGICON()
    ON_BN_CLICKED(IDC_CONNECT, OnConnect)
    ON_BN_CLICKED(IDC_SEND, OnSend)
    ON_MESSAGE(RE_RECEIVED,OnReceive);
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()
```

在 CMyWcDlg 类中，声明函数 OnReceive()，完整的 MyWcDlg.h 文件如下：

```
#include"WCSock.h"
class CMyWcDlg : public CDialog
{
// Construction
public:
        CMyWcDlg(CWnd * pParent = NULL); // standard constructor
        CWCSock sockClient;
        _DATA m_Dat;

        LRESULT OnReceive(WPARAM wparam,LPARAM lParam);
        // Dialog Data
            //{{AFX_DATA(CMyWcDlg)
            enum { IDD = IDD_MYWC_DIALOG };
        CListBox m_ctlSent;
        CListBox m_ctlRecvd;
        int      m_nPort;
        CString  m_server_name;
        CString  m_client_name;
        CString  m_send_info;
        //}}AFX_DATA

        // ClassWizard generated virtual function overrides
        //{{AFX_VIRTUAL(CMyWcDlg)
        protected:
        virtual void DoDataExchange(CDataExchange * pDX); // DDX/DDV support
        //}}AFX_VIRTUAL

        // Implementation
protected:
        HICON m_hIcon;

        // Generated message map functions
        //{{AFX_MSG(CMyWcDlg)
        virtual BOOL OnInitDialog();
        afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
        afx_msg void OnPaint();
```

```
        afx_msg HCURSOR OnQueryDragIcon();
        afx_msg void OnConnect();
        afx_msg void OnSend();
        afx_msg void OnClose();
        //}}AFX_MSG
        DECLARE_MESSAGE_MAP()
};
```

在 CMyWcDlg 类的实现文件中添加函数体：

```
LRESULT CMyWcDlg::OnReceive(WPARAM wparam,LPARAM lParam)
    {
      _DATA *dat;
      dat = new _DATA;
      memcpy(dat,(_DATA *)wparam,sizeof(_DATA));
      CString str = dat->m_strName;
      str += ":";
      str += dat->m_dbData;
      //将接收到的信息添加并显示在列表编辑框
      m_ctlRecvd.AddString(str);
      //显示列表编辑框内容
      UpdateData(false);
      delete dat;
      return 1;
    }
```

### 9.4.9 处理控件的状态更新

在 OnClose()和 OnInitDialog()函数中末添加如下控件的状态更新语句：

```
GetDlgItem(IDC_CONNECT)->EnableWindow(true);
GetDlgItem(IDC_SEND)->EnableWindow(false);
GetDlgItem(IDC_CLOSE)->EnableWindow(false);
GetDlgItem(IDC_SERVER_PORT)->EnableWindow(true);
GetDlgItem(IDC_CLIENT_NAME)->EnableWindow(true);
GetDlgItem(IDC_SERVER_NAME)->EnableWindow(true);
```

在 OnConnect()函数中末添加如下控件的状态更新语句：

```
GetDlgItem(IDC_CONNECT)->EnableWindow(false);
GetDlgItem(IDC_SEND)->EnableWindow(true);
GetDlgItem(IDC_CLOSE)->EnableWindow(true);
GetDlgItem(IDC_SERVER_PORT)->EnableWindow(false);
GetDlgItem(IDC_CLIENT_NAME)->EnableWindow(false);
GetDlgItem(IDC_SERVER_NAME)->EnableWindow(false);
```

### 9.4.10 编译、连接运行

客户端程序开发完毕。编译、连接运行，可以看到图 9.2 所示的用户界面。

### 9.4.11　ClistBox 类

列表框控件提供了文本项的列表，供用户观察和选择，用户通过对话框模板在对话框中加入列表控件，使用 ClassWizard 创建一个定义，作为控件对象 ClistBox 的变量。列表框控件常用的成员函数如表 9.4 所示。

表 9.4　列表框控件常用的成员函数

| 函数名 | 说　明 |
| --- | --- |
| ModifyStyle | 改变列表框的风格 |
| AddString | 向列表框加入项 |
| InsertString | 在指定位置加入一个字符串项 |
| SetTopIndex | 设置某点位于列表框的顶部 |
| DeleteItem | 删除某项 |
| GetCount | 获得总项数 |
| GetCurSel | 获得所选项序号 |
| GetText | 获得指定项文本 |

## 9.5　聊天室服务器端应用程序

### 9.5.1　创建工程 MyWs

创建 MyWs 工程的步骤如下：

（1）启动 Visual C++6.0，从“File”菜单中选择“New”。此时 Visual C++ 将显示一个“New”对话框。

（2）在“New”对话框中选择“Project”标签，然后选择“MFC AppWizard(exe)”类型，告诉 Visual C++ 即将创建一个 EXE 程序。

（3）在“New”对话框的“Project name”文本编辑框中输入“MyWs”，单击位于“Location”框右边的小按钮，再从下拉的对话框中选择“D:\MYVC”目录，使新创建的工程文件放置在“D:\MYVC”目录之下。

（4）单击“New”对话框的“OK”按钮，此时 Visual C++ 将显示“MFC AppWizard - Step 1”对话框。

（5）在“MFC AppWizard - Step 1”对话框中，选择“Dialog based”，创建一个基于对话框的应用程序，然后单击“Next”按钮。

（6）在“MFC AppWizard - Step 2 of 4”对话框中，选择“Windows Sockets”支持。其他接受默认设置，如图 9.11 所示。

（7）然后单击“Finish”按钮，此时 Visual C++ 将显示“NewProject Information”窗口。

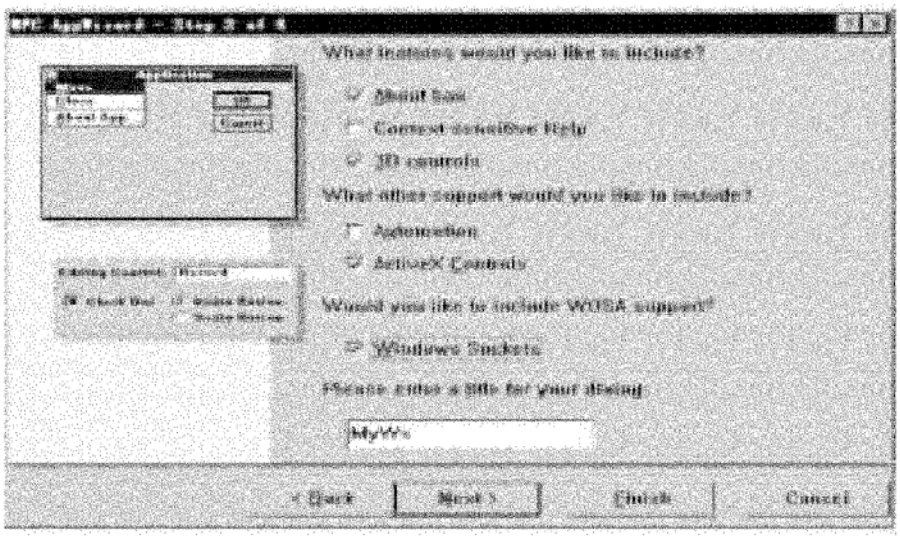



图 9.11 选择“Windows Sockets”支持对话框

(8) 单击“OK”,于是 Visual C++ 就会创建 MyWs 工程以及相关的所有文件。

### 9.5.2 可视化设计

按照图 9.1 用户界面,设计主窗口界面的步骤如下:

(1) 在“Project Workspace”窗口中,单击“ResourceView”标签,把 MyWs resources 扩展开,然后再扩展 Dialog,最后,双击“IDD _ MYWS _ DIALOG”项,Visual C++ 显示出处于设计状态的“IDD _ MYWS _ DIALOG”对话框。

(2) 在“IDD _ MYWC _ DIALOG”对话框中,根据表 9.5 中的控件属性编辑对话框资源,设计完毕后对话框如图 9.12 所示。

表 9.5 控件属性表

| 对 象 | 属 性 | 设 置 |
|---|---|---|
| Button | ID<br>Caption | IDC _ CLOSE<br>关 闭 |
| Button | ID<br>Caption | IDC _ START<br>启 动 |
| Edit Box | ID | IDC _ SERVER _ PORT |
| List Box | ID<br>Selection<br>Vertical Scroll<br>Horizontal Scroll | IDC _ RECEIVED _ INFO<br>Muliline(Styles 标签)<br>Checked(Styles 标签)<br>Checked(Styles 标签) |
| Group Box | ID<br>Caption | IDC _ STATIC<br>请输入以下信息 |

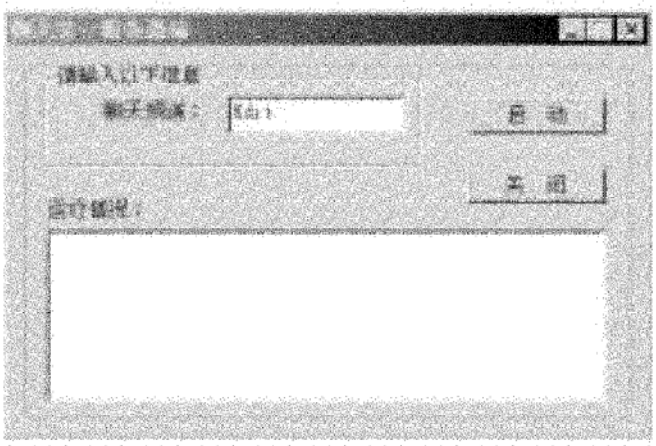


图 9.12　设计完毕后的“IDD _ MYWS _ DIALOC”对话框

### 9.5.3　创建一个侦听类 CLSock

在 MyWs 工程中，创建一个 CSocket 类的派生类 CLSock，用于侦听客户的连接请求。

操作步骤如下：

（1）选择“Insert”菜单的“New Class”菜单项，弹出“New Class”对话框。

（2）在“New Class”对话框中，从“Base class”下拉式列表框中选取“CSocket”作为 CLSocket 的基类，如图 9.13 所示。

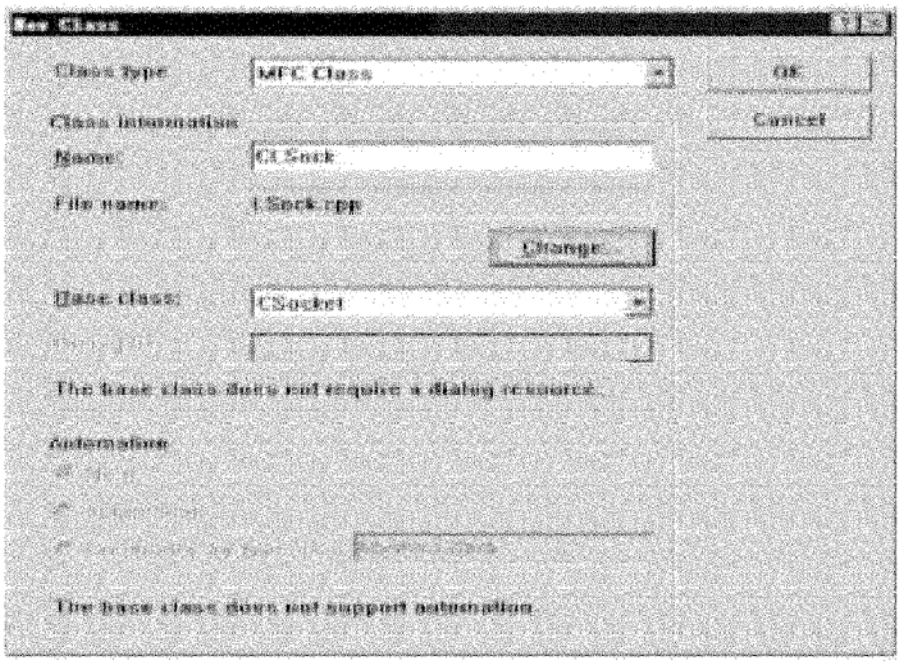



图 9.13　创建新类 CLSock 对话框

（3）单击“OK”按钮，VC ++ 就为 MyWs 工程创建了一个新类 CLSocket，相应地为工程创建了

两个文件 LSocket.h 和 LSocket.cpp。

### 9.5.4 增加一个读/写类 CRWSock

创建 CSocket 类的派生类 CRWSock，专门用于读/写，并与侦听类 CLSock 位于相同的文件 LSock.h 和 LSock.cpp 中。

创建步骤如下：

（1）选择“Insert”菜单的“New Class”菜单项，弹出“New Class”对话框。

（2）在“New Class”对话框中从“Base class”下拉式列表框中选取“CSocket”作为 CRWSock 的基类。

（3）单击“Change...”按钮，弹出“Change Files”对话框。

（4）在“Change Files”对话框中，将“Header file”编辑框修改为“LSock.h”，“Implementation file”编辑框框修改为“LSock.cpp”，如图 9.14 所示。

这样就使新创建的类 CRWSock 与类 CLSock 位于相同的文件 LSock.h 和 LSock.cpp 中。

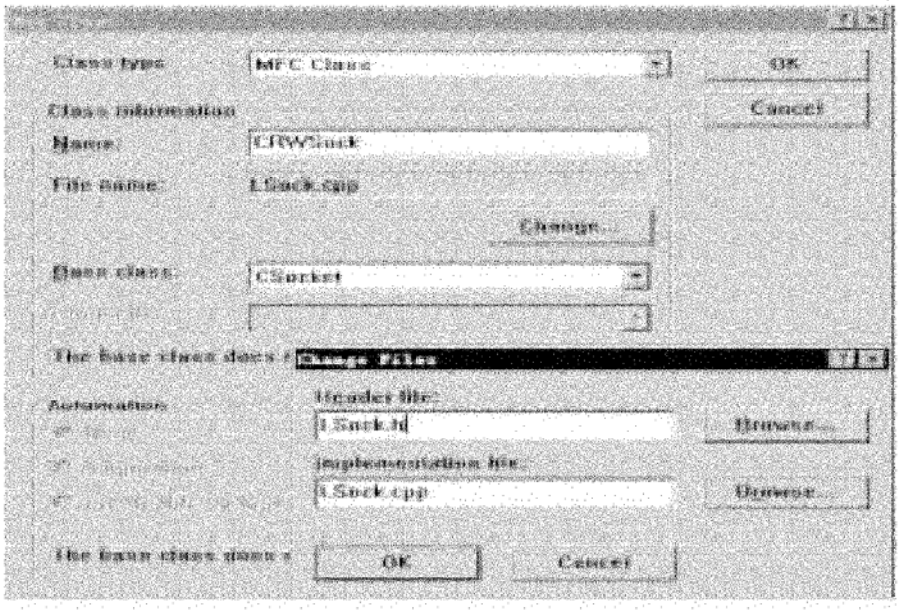



图 9.14 “New Class”和“Change Files”对话框

（5）单击“Change Files”对话框中的“OK”按钮，返回“New Class”对话框。

（6）单击“New Class”对话框中的“OK”按钮，创建 CSocket 类的派生类 CRWSock，，并与侦听类 CLSock 位于相同的文件 LSock.h 和 LSock.cpp 中。

### 9.5.5 为编辑框控件引入变量

为了用编辑框中输入的服务器端口来创建通信端口，将接收到的数据显示在对应的编辑框中，必须为它们引入变量。

用“MFC ClassWizard”，按表 9.6 所列属性为各编辑控件引入关联变量。

表 9.6　各编辑控件的关联变量属性表

| 控件 | ID 标识符 | 变量类型 | 变量名 |
|---|---|---|---|
| List Box | IDC _ RECEIVED _ INFO | CListBox | m _ ctlRecvd |
| Edit Box | IDC _ SERVER _ PORT | UINT | m _ server _ port |

### 9.5.6　修改 CRWSock 和 CLSock 类

**1.** 添加结构体成员变量

为了同客户进行通信，首先需要定义一个通信协议，与客户方保持一致。因此利用一个结构来传递信息，在 LSock.h 文件中添加如下结构：

```
typedef struct
{
   char m_strName[255];      //客户名字
   char m_dbData[255];       //要发送的数据或信息
   BOOL m_bOnline;           //客户是否在线
}_DATA;
```

然后，在 CRWSock 类中添加_ DATA 结构的对象作为成员变量：_ DATA m _ Dat;

**2.** 添加 **CMyWsDlg** 类指针成员变量，修改 **CMyWsDlg** 的构造函数

首先在 CRWSock 类的定义之前添加语句声明：class CMyWsDlg;

再添加 CMyWsDlg 类指针成员变量：CMyWsDlg * m _ p;

```
class CMyWsDlg;
class CRWSock : public CSocket
{
// Attributes
public:
   _DATA m_Dat;
   // Operations
public:
   CRWSock(CMyWsDlg * p);

        virtual ~CRWSock();
   ......
protected:
private:
        CMyWsDlg * m_p;
};

CRWSock::CRWSock(CMyWsDlg * p)
{m_p = p;
}
```

**3.** 在 **CLSock** 类，添加 **CMyWsDlg** 类指针成员变量，修改 **CLSock** 的构造函数

```
class CLSock : public CSocket
{
// Attributes
public:
        ......
// Operations
public:
        CLSock(CMyWsDlg * p);
        virtual ~CLSock();
        ......
protected:
private:
         CMyWsDlg * m_p;
};
CLSock::CLSock(CMyWsDlg * p)
{m_p = p;
}
```

### 9.5.7 修改 CMyWsDlg 类

**1.** 添加头文件

```
#include"LSock.h"
```

**2.** 添加成员变量和成员函数

```
class CMyWsDlg : public CDialog
{
// Construction
public:
        CMyWsDlg(CWnd * pParent = NULL); // standard constructor
        void AcceptClient();
        void ReadMessage(CRWSock * sock);
        _DATA m_Dat;

   // Dialog Data
       //{{AFX_DATA(CMyWsDlg)
        ......
        DECLARE_MESSAGE_MAP()
private:
       void SendMessage(CRWSock * socket,_DATA *buffer);
       // 添加一个 CPtrList 类的对象，用于管理各个客户的套接口
       CPtrList m_list;
       CRWSock * m_clientSocket;
       CLSock * m_listenSocket;
```

```
};
```

**3.** 初始化成员变量

```
CMyWsDlg::CMyWsDlg(CWnd * pParent /* = NULL */)
    : CDialog(CMyWsDlg::IDD, pParent)
{  m_clientSocket = NULL;
   m_listenSocket = NULL;
   //{{AFX_DATA_INIT(CMyWsDlg)
   m_server_port = 800;
   //}}AFX_DATA_INIT
   // Note that LoadIcon does not require a subsequent DestroyIcon in Win32
   m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);
}
```

**4.** 在 **MyWsDlg.cpp** 文件中添加实现函数体

```
void CMyWsDlg::AcceptClient()
{
   m_clientSocket = new CRWSock(this);
   if(! m_listenSocket->Accept(*m_clientSocket))
   {
      AfxMessageBox("请求连接失败");
      delete m_clientSocket;
      m_clientSocket = NULL;
      return;
   }
   m_list.AddTail(m_clientSocket);
}
```

## 9.5.8　处理接收客户的信息

**1.** 添加虚拟函数 **OnReceive()**

如图 9.15 所示，用“MFC ClassWizard”在 CRWSock 类中添加虚拟函数 OnReceive()。

**2.** 编写程序代码

在虚拟函数 OnReceive()中接收客户传来的数据，首先添加到列表编辑框并显示，然后再分两种情况处理：

（1）当收到客户离线信息时，就从套接口链表中删除该结点。

（2）否则，将收到的信息返回客户。

```
void CRWSock::OnReceive(int nErrorCode)
{
   // TODO: Add your specialized code here and/or call the base class
   CSocket::OnReceive(nErrorCode);
   m_p->ReadMessage(this);
```

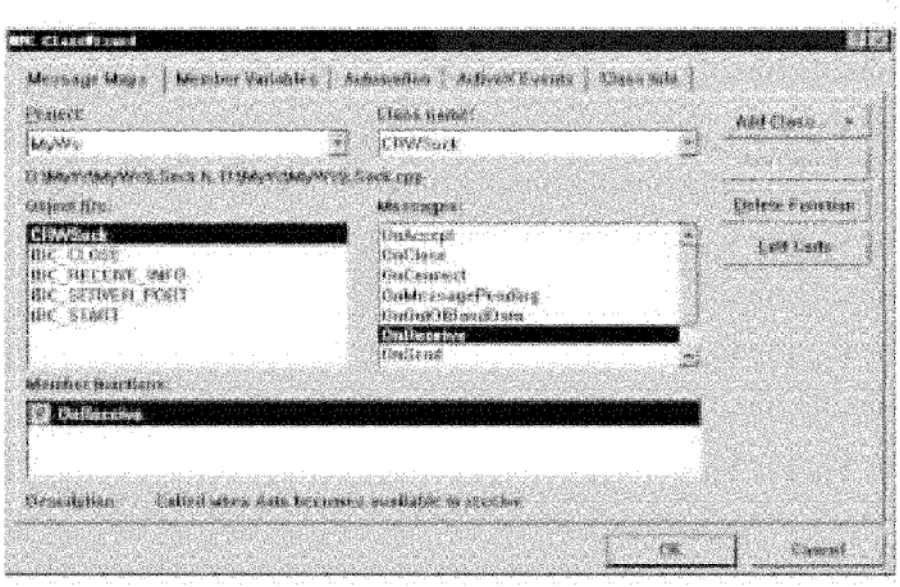


图 9.15 在 CRWSock 类中添加虚拟函数 OnReceive()

```
}
void CMyWsDlg::ReadMessage(CRWSock * sock)
    {
      int len = sock->Receive(&m_Dat,sizeof(m_Dat));
      if(len == sizeof(m_Dat))
      {
        _DATA *dat = &m_Dat;
        CString str = "收到客户机(";
        str += dat->m_strName;
        str += ")的信息:";
        str += dat->m_dbData;
        m_ctlRecvd.AddString(str);
        if(m_Dat.m_bOnline)SendMessage(sock,&m_Dat);
        else //如果客户不在线,则关闭套借口,并从链表中删除
            {
              sock->Close();
              POSITION pos;
              pos = m_list.Find(sock);
              m_list.RemoveAt(pos);
              if(sock != NULL)delete sock;
            }
      }
    }

    void CMyWsDlg::SendMessage(CRWSock * socket,_DATA *buffer)
        {
          POSITION pos = m_list.GetHeadPosition();
          while(pos != NULL)
          {  CRWSock * client =(CRWSock *)m_list.GetNext(pos);
```

```
            if(client == socket)continue;
               client -> Send(&m_Dat,sizeof(m_Dat));
         }
      }
```

### 9.5.9　处理客户的连接请求

按照下面的方法修改 CLSocket 和 CMyWsDlg 类，以便与客户机通信。

（1）在 CmyWsDlg 类中，添加一个 CPtrList 类的对象，用于管理各个客户的套接口：

```
CPtrList m_list;
```

（2）如图 9.16 所示，用“MFC ClassWizard”在 CLSock 类中添加虚拟函数 OnAccept()，用于处理客户的连接请求。

```
void CLSock::OnAccept(int nErrorCode)
{

   CSocket::OnAccept(nErrorCode);
   ASSERT(m_p);
   m_p -> AcceptClient();
}
```

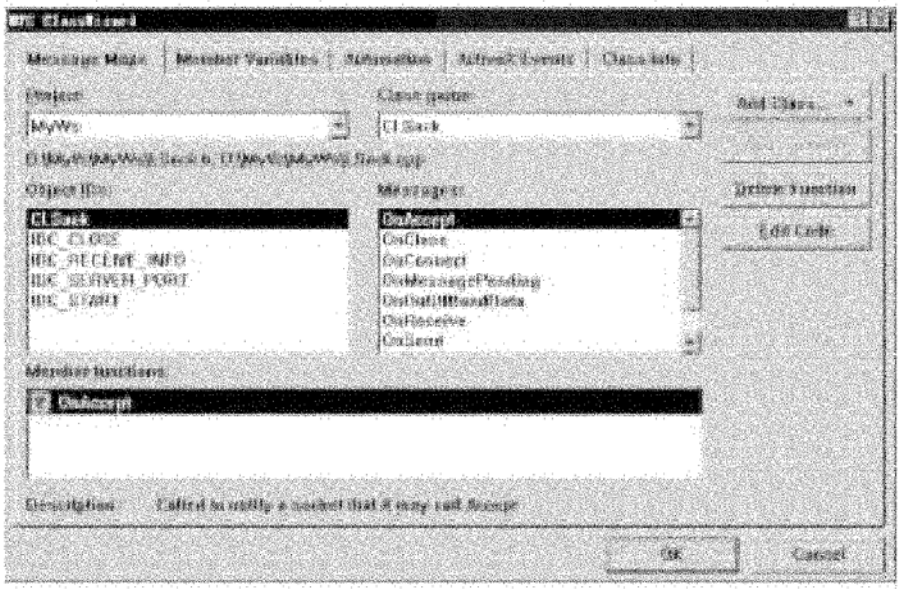


图 9.16　在 CLSock 类中添加虚拟函数 OnAccept()

在该函数中创建一个套接口，并添加到套接口链表中。

```
void CMyWsDlg::AcceptClient()
     {
       m_clientSocket = new CRWSock(this);
       if(! m_listenSocket -> Accept( * m_clientSocket))
```

```
        {
            AfxMessageBox("请求连接失败");
            delete m_clientSocket;
            m_clientSocket = NULL;
            return;
        }
        m_list.AddTail(m_clientSocket);
    }
```

### 9.5.10 为“启动”、“关闭”按钮的 BN_CLICKED 事件编写代码

**1.** 添加消息响应函数

用“MFC ClassWizard”为“IDD_MYWC_DIALOG”对话框中的 Button 控件(IDC_CLOSE 和 IDC_START)的 BN_CLICKED 事件添加消息响应函数：

```
void OnStart()、void OnClose()
```

**2.** 编写程序代码

```
void CMyWsDlg::OnClose()
{
    POSITION pos = m_list.GetHeadPosition();
    while(pos != NULL)
        {
            CRWSock * client = (CRWSock *)m_list.GetNext(pos);
            delete client;
        }
    if(m_listenSocket != NULL)
    {
        m_listenSocket->Close();
        delete m_listenSocket;
    }
}

void CMyWsDlg::OnStart()
{
    // TODO: Add your control notification handler code here
    UpdateData(true);
    m_listenSocket = new CLSock(this);
    if(! m_listenSocket->Create(m_server_port))
    {   AfxMessageBox("创建 Socket 失败!");
        delete m_listenSocket;
        m_listenSocket = NULL;
        return;
    }

    if(! m_listenSocket->Listen())
```

```
        {MessageBox("端口设置失败","网络信息",MB_OK);
        delete m_ListenSocket;
        m_ListenSocket = NULL;
        return;
        }
}
```

### 9.5.11　处理控件的状态更新

在 OnStart()函数中末添加如下控件的状态更新语句：

```
GetDlgItem(IDC_START)->EnableWindow(false);
GetDlgItem(IDC_CLOSE)->EnableWindow(true);
GetDlgItem(IDC_SERVER_PORT)->EnableWindow(false);
```

在 OnClose()和 OnInitDialog()函数中末添加如下控件的状态更新语句：

```
GetDlgItem(IDC_START)->EnableWindow(true);
GetDlgItem(IDC_CLOSE)->EnableWindow(false);
GetDlgItem(IDC_SERVER_PORT)->EnableWindow(true);
```

### 9.5.12　编译、连接并运行

服务器端程序开发完毕。编译、连接并运行，可以看到图 9.1 所示的用户界面。请读者验证本章开始设计的公众聊天室程序目标。

### 9.5.13　CPtrList 类

CPtrList 类用于将某类对象以链表形式进行管理，它提供成员函数实现对链表的基本操作，例如，在 9.5.9 节中，服务器端就是用一个链表来处理客户的连接请求的，用于管理各个客户的套接口。

CPtrList 类型链表的常用成员函数如表 9.7 所示。

表 9.7　CPtrList 类型链表的常用成员函数

| 类　型 | 函数名 | 说　明 |
|---|---|---|
| Head/Tail Access | GetHead(GetTail) | 返回链表的头(尾)元素 |
| Operations | RemoveHead(RemoveTail) | 删除链表的头(尾)元素 |
| | AddHead(AddTail) | 将元素添加到链表的头(尾) |
| | RemoveAll | 删除链表中的所有元素 |
| Iteration | GetHeadPosition(GetTailPosition) | 返回链表的头(尾)元素位置 |
| | GetNext(GetPrev) | 返回链表中后(前)一元素 |
| Retrieval/Modification | GetAt | 返回链表中指定位置的元素 |
| | SetAt | 把某元素链接到链表中的指定位置 |
| | RemoveAt | 在链表中，删除指定位置的元素 |
| Insertion | InsertBefore(InsertAfter) | 将新元素插入到指定位置之前(后) |
| Searching | Find | 在链表中查找，返回给定元素的位置 |

# 习 题 九

1. 修改聊天室程序，使客户断聊天室信息中，也包含自己发送的文字信息；
2. 修改聊天室程序，在客户端，增加客户登入界面。
3. 完善聊天室程序，添加客户可选的聊天对象，即私人聊天或多人聊天。

# 第10章

# 多 线 程

## 本章导读

随着操作系统技术的发展，先后出现了多道程序设计、分时系统、分布式等许多技术，提高了系统运行的效率。用进程的观点来研究操作系统是当今普遍采用的方法，进程、线程概念的出现，对于提高系统并行性具有重要意义。单纯的顺序执行的方式已经不能满足要求，多线程的程序设计也就应运而生。

多线程技术能很好地解决一个应用程序中多个任务的操作。提高了资源的利用率，使某些复杂的并行程序设计简单化。本章将通过编写桌面时差时钟程序，使读者掌握：

- 用 Visual C++6.0 进行多线程的程序设计
- 用不同方式进行线程间的通信
- 线程的同步

---

## 10.1 桌面时差时钟

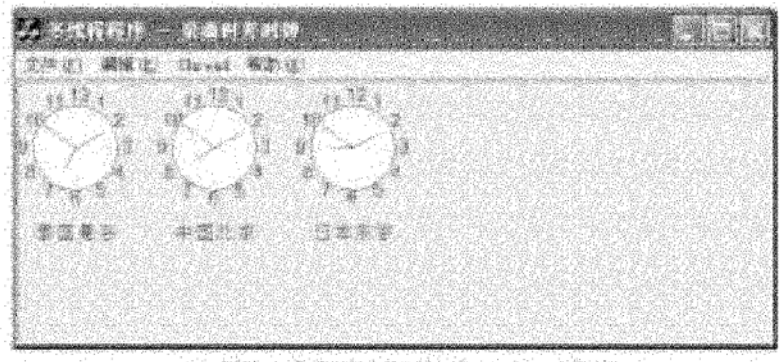
图 10.1(a)所示就是一个基于 MFC 的多线程应用实例。程序主窗口有 3 个活动的时差时钟，用户可以激活第 4 个桌面时钟，也可以终止已激活的时钟。

再激活一个桌面时钟的操作方法是：选择“Thread”菜单下的“Start Thread”菜单项，窗口中将显示新启动的时钟，如图 10.1(b)所示。

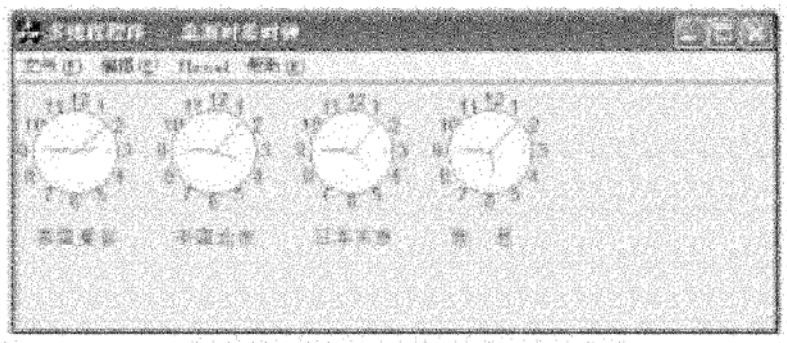
终止已激活的时钟的操作方法是：选择“Thread”菜单下的“End Thread”菜单项，所有桌面上的时钟就会停止运转。

事实上每启动一个桌面时钟，就是要在应用程序中创建一个线程。终止已激活的时钟就是终止线程的执行。

下面将以开发桌面时差时钟程序为实例，讲述线程的创建与终止方法，解决线程间的通信和同步问题，从而，说明如何开发多线程程序。



(a) 创建3个桌面时差时钟的Mythread程序



(b) 创建4个桌面时差时钟的Mythread程序

图 10.1 基于 MFC 的多线程实例

## 10.2 多线程概述

### 10.2.1 多线程与多任务

Windows 95/98/2000/NT 支持两种类型的多任务。第一种类型是基于进程的，进程本质上是一个正在运行的程序，每个进程是由私有的虚拟地址空间、代码、数据和其他各种系统资源组成。进程在运行过程中创建的资源随着进程的终止而被销毁，所使用的系统资源在进程终止时释放或关闭。对于基于进程的多任务而言，多个进程可同时执行。

第二种类型的多任务是基于线程的，线程是进程内的一个执行路线。在 Windows 95/98/2000 中每个进程至少包含一个线程(即主执行线程，它无需用户主动创建，是由系统将应用程序启动后创建的)。用户根据需要可在应用程序中创建其他线程，因此，基于线程的多任务使得同一程序的多个线程并发地运行于同一个进程中(即多个部分可以同时运行)。一个进程中的所有线程都在该进程的虚拟地址空间中，使用这些虚拟地址空间、全局变量和系统资源，所以线程之间的通信要比进程容易得多，多线程的程序设计在实际中的使用也较为广泛。

一个线程很象一个子程序，应用程序能够创建若干可以同时运行的线程，这提高了系统运行的效率。因此，在实际应用中，具有重要意义。

例如，排版打印程序，当用户发出打印命令后，由于文档数量大，在执行打印排版格式等操作可能耗时较长，因此一个设计合理的应用程序不会在每次用户发出打印命令后让用户在那里等待打印完之后才开始工作；而且在打印过程中，用户发现错误后又要及时地终止打印。为此，打印工作可以由一个线程来负责，在后台处理，用户还可以继续做自己的工作，同时线程又能响应用户输入的命令，作出判断，包括终止打印这样的命令。

图 10.1 所示的桌面时差时钟，是一个在 Visual C++6.0 中，使用 MFC 类库开发的多线程实例程序。在该实例中，用户可以创建多个时钟，每个时钟是一个子函数中无限循环执行的结果，是该实例程序的一个线程。

### 10.2.2 线程创建

在 MFC 类库中，每个 CWinThread 对象表示应用程序的一个执行线程。MFC 将线程分为两种类型：用户界面线程(user - interface thread)和工作者线程(worker thread)。

每个进程都必须包含一个称之为主线程的线程，主线程是用户界面线程，在应用程序启动时自动创建和启动。

然而，在同一进程中可创建不止一个的可运行的线程，即工作者线程。通常，一旦创建了新的线程，线程便开始运行。所以，每个进程从某一线程的执行开始进而可创建一个或多个附加线程。这就是基于线程的多任务所支持的多任务方法。

工作者线程的创建分两步：实现控制函数和启动工作者线程。

**1.** 实现控制函数

控制函数用于定义实现线程，例如，在桌面时差时钟实例中，就是控制秒针旋转的程序段。因此，进入控制函数时，线程启动，也即秒针开始旋转；一旦退出控制函数，线程终止，秒针也就停止旋转。控制函数的原型如下：

```
UINT MyConttollingFunction(LPVOID pParam);
```

其中，MyConttollingFunction 是控制函数名称，参数 pParam 是一个 32 位值，它是在创建线程时传给构造函数的值。

**2.** 启动工作者线程

启动工作者线程，就是如何让程序开始执行控制函数，在本章中即如何让秒针开始旋转。其方法是调用全局函数 AfxBeginThread()，其原型如下：

```
CWinThread * AfxBeginThread(
                AFX_THREADPROC pfnThreadPro, LPVOID pParam,
                int nPriority = THREAD_PRIORITY_NORMAL,
                UINT nStackSize = 0,
                DWORD dwCreateFlags = 0,
                LPSECURITY_ATTRIBUTES lpSecurityAttrs = NULL);
```

其中参数 pfnThreadPro，为线程控制函数的名称，参数 pParam 为传递给线程函数 pfnThreadPro()的 32 位参数值，其他参数可以使用默认值。

调用函数 OnStartThread()，就创建一个以 ThreadProc()为线程控制函数的工作者线程。

```
void CMyThreadView::OnStartThread()
{
   //TODO: Add your command handler code here
   HWND hWnd = GetSafeHwnd();
   AfxBeginThread(ThreadProc,hWnd);              // 启动工作者线程
}
```

### 10.2.3 线程终止

线程终止通常有两种情况：一是线程的控制函数执行完毕正常退出，另一种情况是线程被强制终止。例如，在桌面时差时钟实例中，就是让程序退出使秒针旋转的控制函数，从而秒针终止旋转。

线程的正常终止非常简单，只需使用 Return 返回语句（或使用全局函数）退出线程的控制函数，并且返回一个值用以指明退出原因。通常，0 表示成功，其他值表示不同的错误原因。值得注意的是，线程提前终止，只需在线程内部调用函数 AfxEndThread()。这将终止线程的运行，释放线程的堆栈，并从内存中删除线程对象。全局函数 AfxEndThread()原型如下：

```
void AfxEndThread(UINT nExitCode);
```

其中参数 nExitCode 指明线程的退出码。

如果从其他线程来终止线程，必须在线程之间设置通信方式。本章中的线程终止程序，将以终止秒钟旋转为例，以设置通信方式来实现。因此安排在线程通信一节再介绍。

## 10.3 一个简单的多线程程序 MyThread

使用 MFC 创建多线程程序是一件非常容易的事。只需建立一个需要和程序的其他部分并发执行的函数（控制函数），然后调用函数 AfxBeginThread()创建一个新的线程来执行该控制函数。新建的线程在控制函数的执行期间都是活动的，控制函数一旦终止，该线程也就终止。

在本节中，将详细介绍如何创建桌面时差时钟多线程程序 MyThread。

### 10.3.1 创建多线程 MyThread 工程

创建多线程 MyThread 工程的步骤如下：

（1）启动 Visual C++6.0。从“File”菜单中选择“New”。此时，Visual C++将显示一个“New”对话框，如图 10.2 所示。

（2）在“New”对话框中选择“Project”标签。然后选择“MFC AppWizard[exe]”类型，告诉 Visual C++即将创建一个 EXE 程序。

（3）在“New”对话框的“Project name”文本框中输入“MyThread”，在“Location”框中输入“G:\MYVC”目录，使新创建的工程文件放置在“G:\MYVC”目录之下。

（4）单击“New”对话框的“OK”按钮。此时显示“MFC AppWizard – Step 1”对话框。在此对话框中，选择“Single document”，创建一个单文档的应用程序。然后单击“Next”按钮。

（5）在接下来的几个对话框中，除在“MFC AppWizard – Step 4 of 6”对话框中，取消所有的选项（如 Docking Toolbar，Initial status bar，Printing，print preview and 3D controls）外，在其余各步，均接受默认值，按“Next”键；在最后一个对话框“MFC AppWizard – Step 6 of 6”中，单击“Finish”按钮，于是，Visual C++ 就会创建 MyThread 工程以及相关的所有文件。

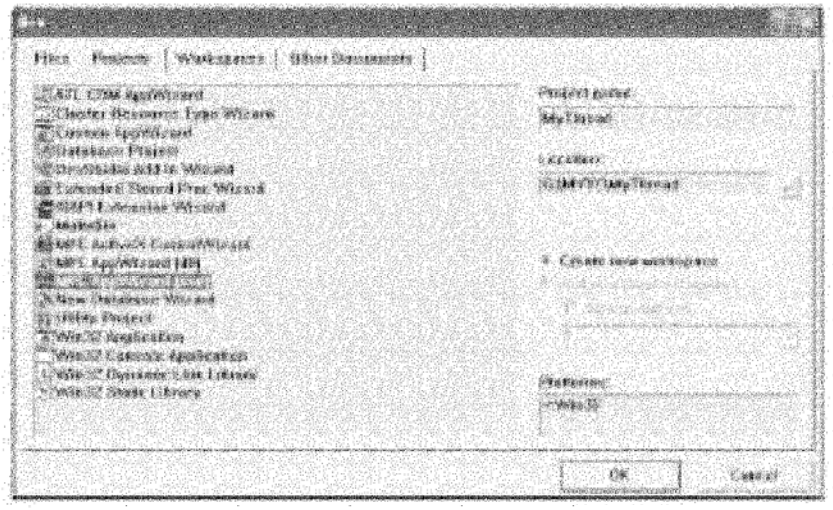


图 10.2　创建多线程 MyThread 工程

### 10.3.2　创建菜单

使用资源编辑器在应用程序的“IDR _ MAINFRAME”菜单增加一个菜单“Thread”，在该菜单下增加菜单项“Start Thread”，ID 号为“ID _ StartThread”，如图 10.3 所示。

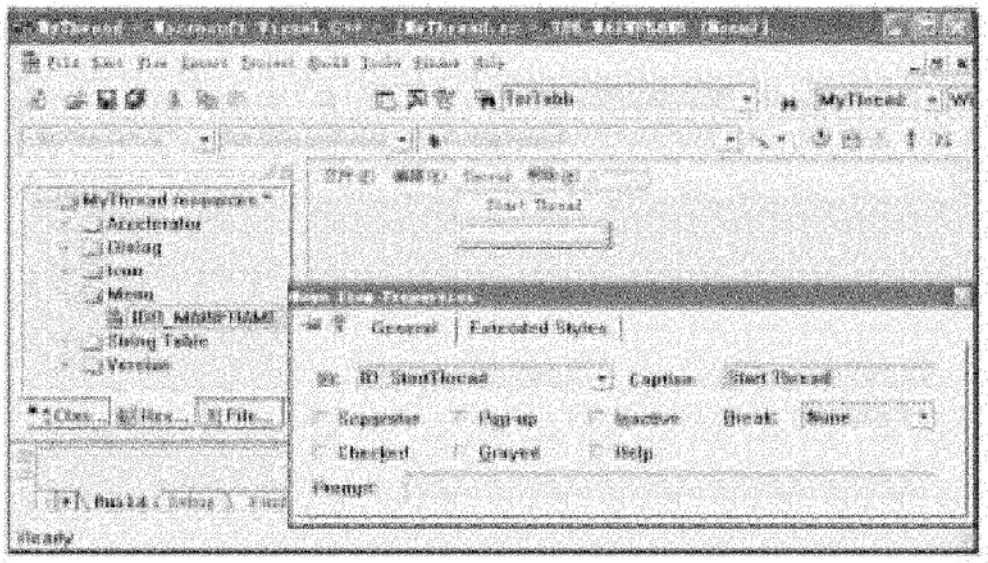


图 10.3　增加菜单与菜单项

### 10.3.3 编写程序代码

**1.** 增加消息响应函数

创建菜单后，接着使用 ClassWizard 来增加菜单项“ID _ StartThread”的消息响应函数 OnStartThread()，如图 10.4 所示。

注意在“Project”框中选择“MyThread”，在类“Class name”框中选择 CMyThreadView，在“Object IDs:”框中选择“ID _ StartThread”。

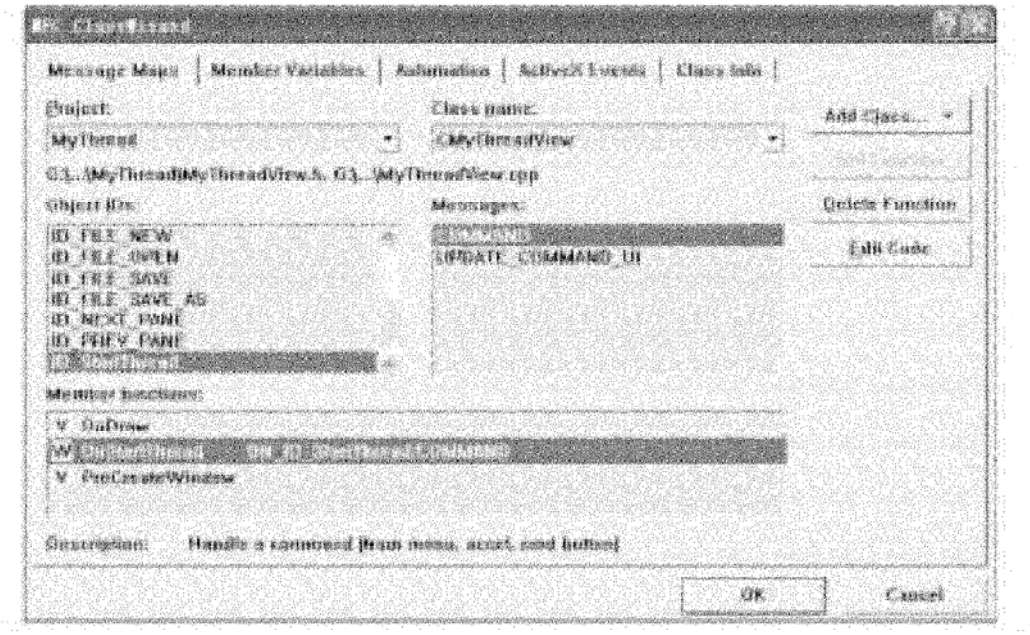



图 10.4 增加菜单项“ID _ StartThread”的消息响应函数

**2.** 编辑消息响应函数代码

在图 10.4 中，单击“Edit Code”按钮，或者直接打开 MyThreadView.cpp 文件，编辑消息响应函数 OnStartThread()，代码如下：

```
void CMyThreadView::OnStartThread()
{
//TODO: Add your command handler code here
HWND hWnd = GetSafeHwnd();
AfxBeginThread(ThreadProc,hWnd);
}
```

在函数 OnStartThread()中，调用全局函数 AfxBeginThread()来创建一个新的线程，ThreadProc 为线程控制函数的名称，参数 hWnd 为传递给函数 ThreadProc()的参数。

**3.** 编辑线程控制函数 **ThreadProc()**

线程控制函数 ThreadProc()，是一个全局函数，而不是类 CMyThreadView 的成员函数。将函数 ThreadProc()放在 OnStartThread()的前面，函数 ThreadProc()的代码如下：

```
// CMyThreadView message handlers
UINT ThreadProc(LPVOID param)
{
   int i = 0;
         double r = 35.0;
   double alpha = 3.14/30.0;     //表针一次走动所旋转的弧度数
   int cx, cy;                   //表针的圆心

   //限制最多启动 8 个线程
   if(ThreadCount >= 8){MessageBeep((WORD)-1);return 0;}
      //确定各表针的圆心的坐标
   cx = 50 + (ThreadCount % 4) * 120;
   if( ThreadCount < 4) cy = 60;
   else                 cy = 200;

   ThreadCount++;          //线程数目计算增加 1
   CClientDC dc(((CFrameWnd*)(AfxGetApp()->m_pMainWnd))->GetActiveView());

   //draw second time begins
   dc.SetROP2(R2_NOT);
   dc.MoveTo(cx,cy);
   dc.LineTo(int(cx + r*cos(i*alpha)),int(cy + r*sin(i*alpha)));     //画一条秒针
   直线

   for( ; ; )              //无限循环,使秒针不停地旋转
   {
      dc.LineTo(cx,cy);    //删除上一条秒针直线
      ;// 画一条秒针直线
      dc.LineTo(int(cx + r*cos(i*alpha)), int(cy + r*sin(i*alpha)))i++;
      if(i==60) i=0;


      ::Sleep(500);        //延迟 500 μs
   }
   return 0;
}                          //End Of ThreadProc
```

函数 ThreadProc()中用到了数学函数 cos() 和 sin(),需要在文件 MyThreadView.cpp 的头部包含头文件 math.h,即加上语句:

```
#include"math.h"
```

特别注意:

(1)“#include"math.h""语句要放在“#include"stdafx.h""语句的下面。

(2)函数 ThreadProc()还用到一个全局变量 ThreadCount,为此需要在文件 MyThreadView.cpp 的头部定义该变量,在 #endif 的后面加上“volatile int ThreadCount;”语句。

(3)创建多线程程序的关键代码,是线程控制函数。在桌面时钟 MyThread 实例中,控制函数 ThreadProc()用直线摸拟秒表的指针,通过不断地绘制直线来摸拟秒表的走动。

函数首先构造一个 CDC 对象,如下所示:

```
CClientDC dc(((CFrameWnd * )(AfxGetApp()->m_pMainWnd))->GetActiveView());
```

在模拟秒表的走动时，首先删除上一条直线，然后绘制新的直线。SetROP2(R2_NOT)用于设置作图模式为：将屏幕当前颜色求反。因此，在某一位置画一条直线时，它是可见的，但在第二次绘制直线时，就不可见了。

**4.** 查看结果

编译、连接 MyThread 程序，在显示的界面上选择执行菜单命令“Thread”，再单击“Start Thread”项，程序将创建一个新的线程，即一个秒针不断旋转的时钟。

可多次选择执行菜单命令“Thread”，再单击“Start Thread”项，每次都将创建一个新的线程，各线程并发执行，如图 10.5 所示。

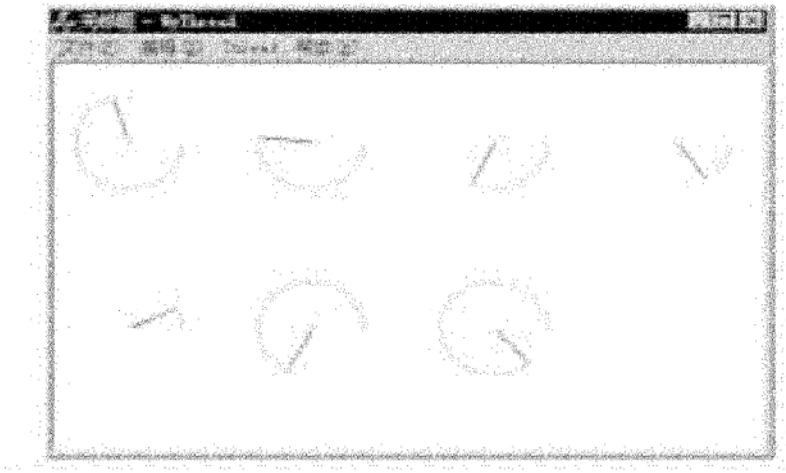


图 10.5 启动 7 个线程

## 10.4 线程间的通信

线程之间通信是经常需要用到的。例如，利用其他线程来终止线程，就必须在线程之间设置通信方式。下面，就在前节创建的时钟多线程实例的基础上，编辑修改程序代码，分别利用线程之间的两种通信方法，以实现退出控制函数，终止线程，使时钟停止运转。

### 10.4.1 使用全局变量进行线程通信

**1.** 创建菜单

首先，使用资源编辑器，给菜单“Thread”增加一个菜单项“Stop Thread”，ID 号为“ID_StopThread”，如图 10.6 所示。

**2.** 增加消息处理函数

使用“MFC ClassWizard”来增加菜单项“ID_StopThread”的消息处理函数 OnStopThread()，如图

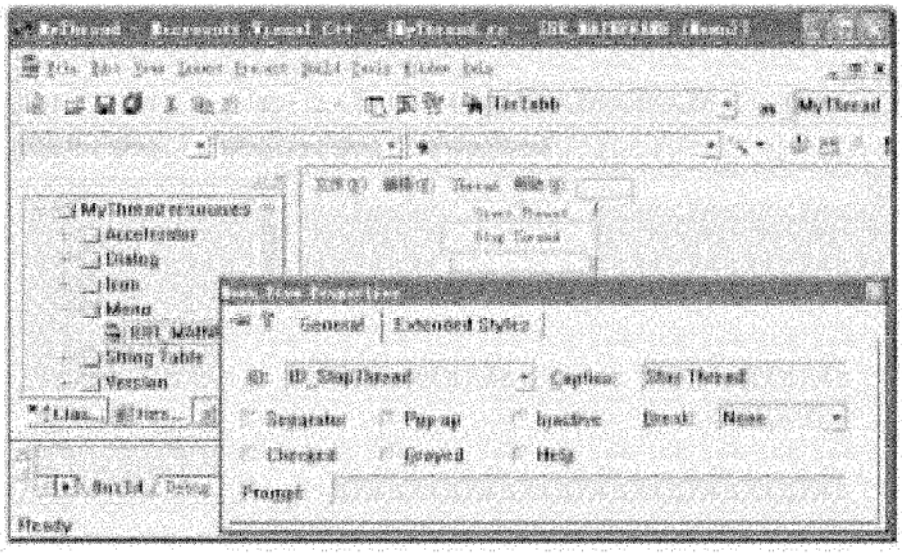


图 10.6 增加菜单项“Stop Thread”

10.7 所示。注意在“Project”框中选择“MyThread”，在类“Class name”框中选择“MyCThreadView”。

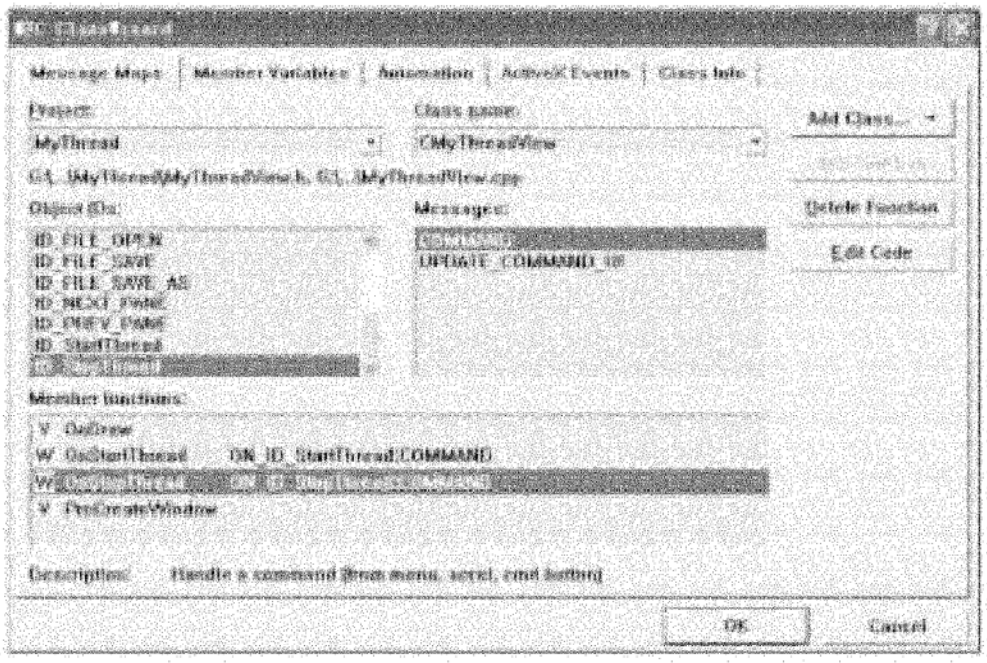


图 10.7 增加菜单项消息处理函数 OnStopThread()

**3.** 编辑消息处理函数代码

在图 10.7 中，单击“Edit Code”按钮，或者直接打开 MyThreadview.cpp 文件，编辑消息处理函数 OnStopThread()，在该函数中增加一条语句 ThreadFlag = 0。

```
Void CMyThreadView:: OnStopThread()
{
```

```
    //TODO: Add your command handler code here
        OnStopThreadFlag = 0;     //设置线程标记为终止
}
```

其中 ThreadFlag 为全局变量，需要在文件 MyThreadView.ccp 的头部予以定义。即增加一行语句：

```
volatile   int OnStopThreadFlag;
```

同时，在函数 OnStartThread()中增加一条语句 ThreadFlag = 1。

```
void CMyThreadView::OnStartThread()
{
   //TODO: Add your command handler code here
   OnStopThreadFlag = 1;
   HWND hWnd = GetSafeHwnd();
   AfxBeginThread(ThreadProc,hWnd);
}
```

**4.** 修改线程控制函数 **ThreadProc()**

在线程控制函数 ThreadProc()内部，OnStop 对 ThreadFlag 全局变量进行监视，如果 ThreadFlag 的值为 0，则终止线程。为此，在 for 无限循环内增加一条 IF 条件判断语句，如果 OnStopThreadFlag 的值为 0，则退出 for 循环来终止线程。

```
UINT ThreadProc(LPVOID param)
{
   ……
   for( ; ; )
   {
      //监视全局变量 ThreadFlag，若值为零则结束线程
      if(ThreadFlag = = 0) break;
      ……
   }
   return 0;
}
```

**5.** 测试结果

编译连接运行程序 MyThread 后，就可以进行测试。首先两次执行菜单命令“Thread”→“Start Thread”，创建两个子线程，与图 10.5 类似。2 个时钟在转动，再执行菜单命令“Thread”→“Stop Thread”，2 个时钟都停止动转，这表明两个线程都终止了。

### 10.4.2 使用自定义消息进行线程通信

显然，使用全局变量进行线程通信有些笨拙，而且有时是有危险的。相比之下，通过发送消息进行线程通信更加安全可靠，更加灵活。只需要自定义一个用户消息，在子线程中就可以通过发送消息给主线程来完成线程间的通信。为此，在使用全局变量进行线程通信的程序的基础上，需按下列步骤，修改程序代码。

**1.** 首先定义一个用户消息

在头文件 MyThreadView.h 的头部，CMyThreadView 类定义之前加入一条定义一个用户消息的语句：

```
const WM_THREADENDED = WM_USER + 100;
```

在 AFX_MSG 消息处理函数声明的//}}AFX_MSG 的后面增加下面一行：

```
afx_msg LONG OnThreadEnded(WPARAM wParam,LPARAM lParam);
```

即在文件 MyThreadView.h 中增加两条语句：

```
//MyThreadView.h                                    //新增加语句
……
const WM_THREADENDED = WM_USER + 100;     //新增加的语句
class CMyThreadView : public CView
{
protected: // create from serialization only
    CMyThreadView();
    DECLARE_DYNCREATE(CMyThreadView)
……
protected:
    //{{AFX_MSG(CMyThreadView)
    afx_msg void OnStartThread();
    afx_msg void OnStopThread();
    //}}AFX_MSG
    afx_msg LONG OnThreadEnded(WPARAM wParam, LPARAM lParam);
    DECLARE_MESSAGE_MAP()
};
……
```

**2.** 消息映射

打开文件 MyThreadView.cpp，在类 CMyThreadView 的消息映射中的//}}AFX_MSG_MAP 的后面增加一行消息映射代码：

```
//MyThreadView.cpp
……
volatile int ThreadCount = 0;
volatile int ThreadFlag;
////////////////////////////////////////////////
// CMyThreadView
IMPLEMENT_DYNCREATE(CMyThreadView, CView)

BEGIN_MESSAGE_MAP(CMyThreadView, CView)
   //{{AFX_MSG_MAP(CMyThreadView)
   ON_COMMAND(ID_StartThread, OnStartThread)
   ON_COMMAND(ID_StopThread, OnStopThread)
   //}}AFX_MSG_MAP
```

```
    ON_MESSAGE(WM_THREADENDED,OnThreadEnded)     //新增加的消息映射语句
END_MESSAGE_MAP()
……
```

**3.** 调用函数::**PostMessage**()发送消息

在线程控制函数 ThreadProc()中，调用 PostMessage 来发送消息 WM_THREADENDED。即在线程控制函数中，添加一条语句：

```
UINT ThreadProc(LPVOID param)
{
    ……
    for( ; ; )
    {
        ……
    }
//发送消息 WM_THREADENDED
::PostMessage((HWND)param,WM_THREADENDED,0,0);  //添加的语句
return 0;
}
```

**4.** 手工编写函数 **OnThreadEnded**

在文件 MyThreadView.cpp 的最后，手工编写函数 OnThreadEnded()，并声明为 CMyThreadView 类的成员函数。

```
LONG CMyThreadView::OnThreadEnded(WPARAM wParam,LPARAM lParam)
{
    CString str;
    str.Format("%s %d %s","第",ThreadCount--,"个线程终止");
    MessageBox(str);
    return 0;
}
```

**5.** 测试结果

完成上面编辑修改后，编译、连接运行程序 MyThread，进行测试。连续 3 次执行菜单命令“Thread”→“Start Thread”，创建 3 个新的线程，这时，时钟在转动，然后选取执行菜单命令“Thread”→“Stop Thread”，可以看到时钟停止转动，同时弹出一个消息对话框显示线程终止，如图 10.8 所示。

图 **10.8** 线程终止

需注意的是，该对话框是从主线程中显示的(子线程发送消息，主线程接到消息后显示消息对话框)，而不是在子线程中显示的，所以，在弹出第一个消息对话框显示第 1 个线

程终止时，所有的时钟均已停止转动。点击“确定”按钮，将弹出第二个消息对话框显示第2个线程终止……等等。一般地，若创建多个新的线程，线程终止时则弹出等同个对话框显示各自线程终止。

### 10.4.3　完善 MyThread 程序

**1. 修改线程控制函数 ThreadProc**

观察图10.1与图10.5，不难发现，除了还没有考虑与实际时间一致外，程序 MyThread 与目标程序有如下差别：

（1）缺少时针和分针。

（2）缺少时钟的数字标识等。

为此，下面将进一步完善 MyThread 程序，除了解决正确的时钟显示外，还要为 MyThread 程序添加时针和分针以及时钟的数字等。

特别需要说明的是，这里是以机器时间为参考，修改后的完整的线程控制函数如下：

```
UINT ThreadProc(LPVOID param)
{
    char currenttime[30]; struct tm * newtime; long ltime;
    time(&ltime);
    newtime = localtime(&ltime);
    strcpy(currenttime,asctime(newtime));
    currenttime[24] = ' '; currenttime[25] = 0;

    int i = 0,k = -15 + newtime -> tm_min;
    double r = 35.0,rk = 28.0,rh = 20.0,rp = 30.0;
    double alpha = 3.14/30.0;
    int cx,cy,r1 = 37;
    cx = 50 + (ThreadCount % 4) * 120;

    if(ThreadCount >= 8){MessageBeep((WORD) -1);return 0;}
    if(ThreadCount < 4)cy = 60;
    else               cy = 200;

    CClientDC dc(((CFrameWnd *)(AfxGetApp() -> m_pMainWnd)) -> GetActiveView
    ());

    //begin draw number
    dc.SetBkMode(TRANSPARENT);
    CFont MyFont;
    MyFont.CreateFont(20,0,0,0,10,false,false,
        0,ANSI_CHARSET,
        OUT_DEFAULT_PRECIS,
        CLIP_DEFAULT_PRECIS,
        DEFAULT_QUALITY,
        DEFAULT_PITCH FF_SWISS,
        "Arial");
    CFont *pOldFont = dc.SelectObject(&MyFont);
    dc.SetTextColor(RGB(0,25,0));
```

```
//显示数字：1,2,……,12
char s[6];
int number = 1,j;
float r2 = 45.0;
for(j = -60;j < 60 - 10;j += 5,number++)
    dc.TextOut(int(cx - 5 + r2 * cos(j * alpha)),
           int(cy - 10 - r2 * sin(j * alpha)),itoa(number,s,10));
//显示时钟地域
switch(ThreadCount)
{
case 0:
     dc.TextOut(cx - 35,cy + int(r2) + 20,"泰国曼谷");
     break;
case 1:
     dc.TextOut(cx - 35,cy + int(r2) + 20,"中国北京");
     break;
case 2:
     dc.TextOut(cx - 35,cy + int(r2) + 20,"日本东京");
     break;
default:
     dc.TextOut(cx - 35,cy + int(r2) + 20,"悉 尼");
     break;
}
dc.SelectObject(pOldFont);

//显示时钟外圆和小圆点
CPen mypen;
mypen.CreatePen(PS_SOLID,1,RGB(20,20,200));
CPen *po;
po = dc.SelectObject(&mypen);
CRect myrect(cx - r1,cy - r1,cx + r1,cy + r1);
dc.Ellipse(&myrect);
for(i = 0;i < 60;i += 5)
{
     dc.MoveTo(int(cx - (rp - 1) * cos(i * alpha)),
           int(cy + (rp - 1) * sin(i * alpha)));
     dc.LineTo(int(cx - (rp + 1) * cos(i * alpha)),
           int(cy + (rp + 1) * sin(i * alpha)));
}
dc.SelectObject(po);
//draw circle end

//画时钟的时针、分针和秒针
dc.SetROP2(R2_NOT);
time(&ltime);
newtime = localtime(&ltime);
s = -15 + newtime->tm_sec;
k = -15 + newtime->tm_min;

switch(ThreadCount)
```

```
{
    case 0:
        newtime->tm_hour = (newtime->tm_hour-1)%24;break;
    case 1:
        newtime->tm_hour = (newtime->tm_hour)%24;break;
    case 2:
        newtime->tm_hour = (newtime->tm_hour+1)%24;break;
    default:
        newtime->tm_hour = (newtime->tm_hour+2)%24;
}
ThreadCount++;
int hour = -15+(newtime->tm_hour)*5+newtime->tm_min/12;
dc.MoveTo(cx,cy);
dc.LineTo(int(cx+rh*cos(hour*alpha)),
      int(cy+rh*sin(hour*alpha)));          //画一条直线 hour
dc.MoveTo(cx,cy);
dc.LineTo(int(cx+rk*cos(k*alpha)),
      int(cy+rk*sin(k*alpha)));             //画一条直线 minute
dc.SetTextColor(RGB(255,0,0));
dc.MoveTo(cx,cy);
dc.LineTo(int(cx+r*cos(i*alpha)),
      int(cy+r*sin(i*alpha)));              //画一条新直线 second
for( ; ; )
{
    if(ThreadFlag==0)break;                 //线程终止
    dc.MoveTo(cx,cy);
    dc.LineTo(int(cx-r*cos(i*alpha)),
          int(cy+r*sin(i*alpha)));          //删除上一条直线
    i++;
    if(i==60-15)
    {
        i=-15;
        dc.SetTextColor(RGB(0,25,0));
        dc.MoveTo(cx,cy);
        dc.LineTo(int(cx+rk*cos(k*alpha)),
            int(cy+rk*sin(k*alpha)));       //删除上一条直线 minute
        k+=1;
        dc.MoveTo(cx,cy);
        dc.LineTo(int(cx+rk*cos(k*alpha)),
              int(cy-rk*sin(k*alpha)));     //画一条新直线
        dc.SetTextColor(RGB(255,0,0));
    }

    dc.MoveTo(cx,cy);
    dc.LineTo(int(cx-r*cos(i*alpha)),
          int(cy+r*sin(i*alpha)));          //画一条新直线
    ::Sleep(1000);
}
::PostMessage((HWND)param,WM_THREADFINISHED,0,0);
```

```
    return 0;
}
```

**2.** 查看结果

编译、连接运行程序 MyThread，选择执行菜单命令“Thread”➤“Start Thread”，程序将创建一个新的线程，即一个秒针不断旋转的时钟。可多次执行菜单命令“Thread”➤“Start Thread”，每次都将创建一个新的线程，各线程并发执行，如图 10.1 所示。

# 10.5 线程同步

## 10.5.1 线程同步概述

对于多线程程序，由于多个线程并发执行，一个应用程序的两个线程可能会同时访问同一数据资源，这时，可能会引起问题。

例如，当一个线程在写某文件的过程中，同时，另外一个线程试图读该文件。读线程不知道该读哪一个版本的数据，这时就可能会读到坏的数据。

又如，我们设想以下函数 ShareThreadFunction()中的程序段：

```
//两个线程共有的线程控制函数 ShareThreadFunction()
void ShareThreadFunction()
{
    ……
    count - - ;        // 全局变量，计数器值递增一
    cout < < count; // 打印新的计数器值
    ……
}
```

是两个线程都要执行的一段递增一个全局计数器的值，然后打印新的计数器值。本来希望打印的是从 1 开始的连续正数。但由于两个线程是并发执行的，在计数和打印两个不同时间片中，两个线程可能出现交叉接管控制权，这样打印的数就不连续了，即可能发生下列情况：

第一个线程递增计数器 Count 的值；第二个线程抢先接到控制，第二个线程递增计数器 Count 的值，同时打印新值；第一个线程再次接收到控制，打印计数器 Count 的值（和第二个线程刚打印的值相同），而跳过了它应当打印的值。

凡此种种，便产生了线程同步问题。线程同步主要出现在下列情况：

对资源的共享访问、线程等待事件和死锁。

下面将介绍解决这类问题的三种基本线程同步方法。

## 10.5.2 使用临界区对象进行线程同步

使用临界区对象进行线程同步，很容易确保在同一时刻，只有一个线程访问同一数据对象。其方法是使用临界区对象，将一段代码放入临界区，制止多于一个线程同时进入临界区中。

在 MFC 应用程序中创建临界区对象很简单，只需创建一个类的实例，即：

```
CCriticalSection criSect;
```

当应用程序要存取需要保护的数据时，只需调用临界区对象的成员函数 Lock()，即：

```
criSect.Lock();
```

存取完毕再调用 Unlock()函数，释放该线程对临界区的控制权，即：

```
criSect.Unlock();
```

下面，将以前面引用的实例：递增和打印计数器值，说明使用临界区进行线程同步的方法。操作步骤如下：

**1.** 创建 **MySyn1** 工程

用 10.3.1 节中完全类似的方法创建一个新工程，类型为 MFC AppWizard(exe)，工程名称为 MySyn1。

**2.** 添加菜单和菜单项

使用资源编辑器在应用程序的“IDR _ MAINFRAME”菜单中，增加一个菜单“Thread”，在该菜单下增加两条菜单项“Synchronize”和“Non _ Synchro”，ID 号分别为“ID _ Synchro”和“ID _ Nonsyn”，如图 10.9 所示。

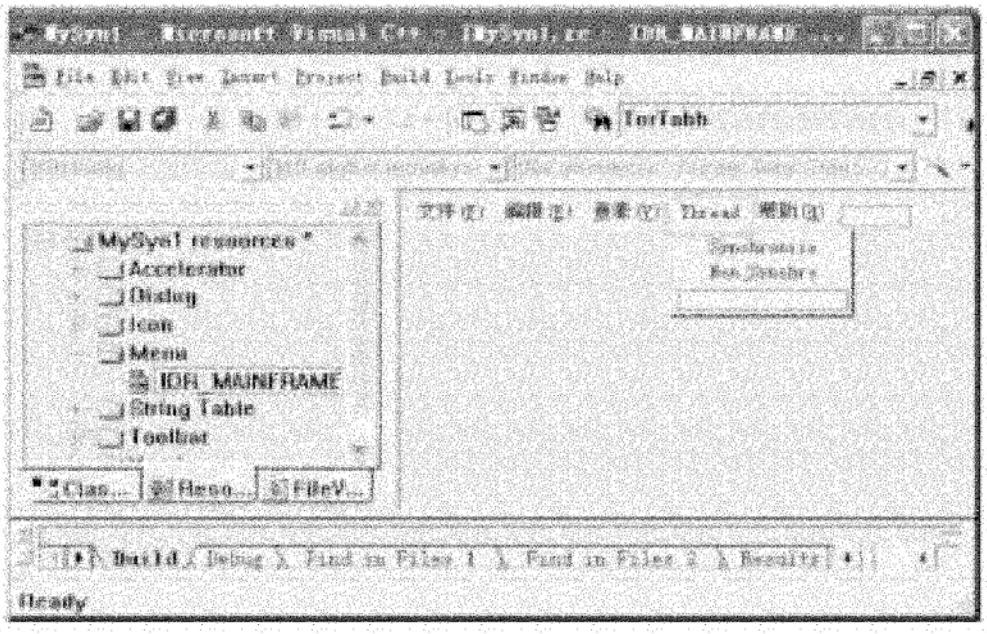



图 10.9　添加菜单和菜单项

**3.** 编写程序代码

创建菜单后，接着使用“MFC ClassWizard”来增加命令“ID _ Synchro”和“ID _ Nonsyn”的命令处

理函数 OnSynchro( )和 OnNonsyn( ),如图 10.10 所示。

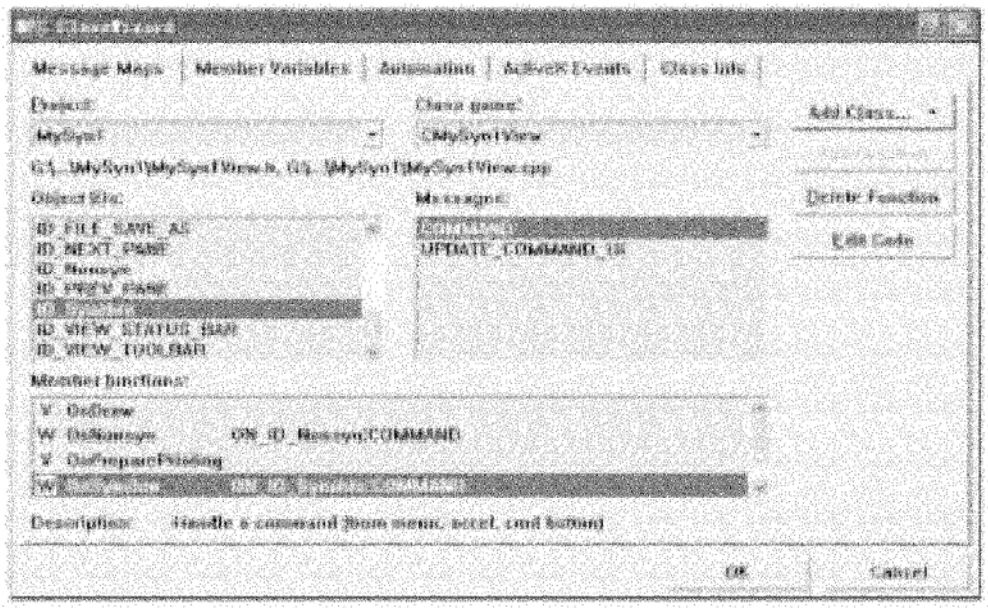


图 10.10 增加菜单项的命令处理函数

编辑函数 OnSynchro( )和 OnNonsyn( )代码,并编辑相应的线程控制函数,最后,需要在文件 MySyn1View.cpp 的头部包含文件 afxmt.h。文件 MySyn1View.cpp 的代码如下:

```
// MySyn1View.cpp 文件的代码如下:
#include"afxmt.h"                          //关键代码行
……
//全局变量
int count = 0;                             //关键代码行
CCriticalSection critical;                 //关键代码行
……
// CMySyn1View message handlers
UINT NonsynFunction(LPVOID param)          //非同步线程控制函数
{    int Data[5], i;
     for(i = 0;i < 3;i ++ )
     {    count ++ ;
          ::Sleep(50);
          Data[i] = count;
     }
     char str[50];
     str[0] = 0;
     for(i = 0;i < 3;i ++ )
     {    int len = strlen(str);
          wsprintf(&str[len],"%d",Data[i]);
     }
     ::MessageBox((HWND)param,str,"Nonsyn Thread",MB_OK);
     return 0;
```

```
}

UINT SynchroFunction(LPVOID param)    // 同步线程控制函数
{    int Data[5], i;
     critical.Lock();                 //保护临界区的数据
     for(i=0;i<3;i++)
     {    count++;
          ::Sleep(50);
          Data[i]=count;
     }
     char str[50];
     str[0]=0;
     for(i=0;i<3;i++)
     {    int len=strlen(str);
          wsprintf(&str[len],"%d",Data[i]);
     }
     critical.Unlock();               //释放该线程对临界区的控制权
     ::MessageBox((HWND)param,str,"synchro Thread",MB_OK);
     return 0;
}

void CMySyn1View::OnNonsyn()          //非同步方式启动三个线程
{
     // TODO: Add your command handler code here
     count=0;
     HWND hWnd=GetSafeHwnd();
     AfxBeginThread(NonsynFunction,hWnd);
     AfxBeginThread(NonsynFunction,hWnd);
     AfxBeginThread(NonsynFunction,hWnd);
}

void CMySyn1View::OnSynchro()         //同步方式启动三个线程
{
     // TODO: Add your command handler code here
     count=0;
     HWND hWnd=GetSafeHwnd();
     AfxBeginThread(SynchroFunction,hWnd);
     AfxBeginThread(SynchroFunction,hWnd);
     AfxBeginThread(SynchroFunction,hWnd);
}
```

**4.** 查看结果

完成上面的工作之后，编译、连接和运行程序 MySyn1，查看运行情况。图 10.11 所示是执行菜单“Thread”→“Synchronize”后的结果，图 10.12 所示是执行菜单“Thread/Non _ Synchro”后的结果。显然，使用临界区进行线程同步之后，3 个线程打印的计数器的值，是均匀递增加的，说明 3 个线程没有交替接管制权，确保在同一时刻，只有一个线程访问同一数据对象 count；而没有进行同步的 3 个线程打印的计数器的值是忽大忽小没有规律的。

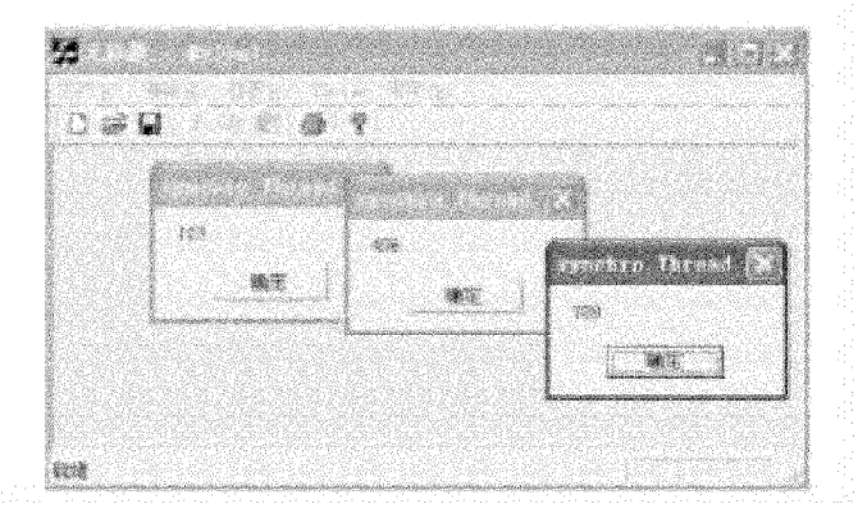


图 10.11 同步线程的运行结果

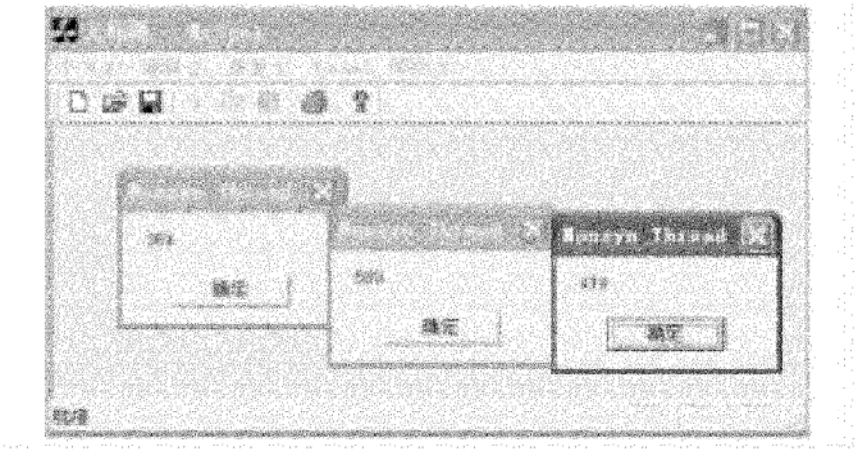


图 10.12 非同步线程的运行结果

### 10.5.3 使用互斥对象（Mutexse）进行线程同步

互斥对象（Mutexse）类似于临界区对象，但比临界区对象更复杂，更具有实际意义。它不仅允许同一程序的不同线程安全共享资源，而且允许不同应用程序的线程安全共享资源。

一个互斥对象只能被一个线程拥有，如果另一线程已经占用了互斥对象，则系统将挂起当前的调用线程，直到这个互斥对象被占用线程释放为止，这时，等待的线程被唤醒并取得对互斥对象的控制。

这里，仅讨论同一应用程序的不同线程安全共享资源的方法，并仍以上面的例子为例，说明如何使用互斥对象进行线程同步。其方法是，当应用程序要存取需要保护的数据时，可以创建一个 CSingleLock 对象，即：

```
CSingleLock singleLock(&mutex);
```

要获得互斥对象的访问，则调用 CSingleLock 对象的成员函数 Lock()，即

```
singleLock.Lock();
```

释放互斥时对象，调用 CSingleLock 对象的成员函数 UnLock()，即

```
singleLock.Unlock();
```

因此，只要对上面例子程序中的线程控制函数稍作改动，同时在文件 MySyn1View.cpp 中创建一个 CMutex 类的实例：

```
CMutex mutex;
```

即可使用互斥对象进行线程同步。

下面列出改动后的代码，编译连接执行，可以得到上面同样结果。

```
// MySyn1View.cpp 文件的代码如下：
#include"afxmt.h"
……
//全局变量
int count = 0;
CMutex mutex;
……

UINT SynchroFunction(LPVOID param)
{
   CSingleLock singleLock(&mutex);     // 用互斥对象保护数据
   singleLock.Lock();
   int Data[5],i;
   for(i = 0;i < 3;i++)
   {    count++;
        ::Sleep(50);
        Data[i] = count;
   }
   char str[50];
   str[0] = 0;
   for(i = 0;i < 3;i++)
   {    int len = strlen(str);
        wsprintf(&str[len],"%d",Data[i]);
   }
   singleLock.Unlock();
   ::MessageBox((HWND)param,str,"synchro Thread",MB_OK);
   return 0;
}
```

### 10.5.4 使用信号量(Semaphores)对象进行线程同步

临界区对象和互斥对象在同一时刻都只允许一个线程存取资源，而信号量对象可以允许多

个线程同时存取资源。

当创建信号量对象时，可以设置访问计数器的值来控制同时使用资源的线程数。每当有一个线程访问资源时，访问计数器的值减一，当访问计数器的值为 0 时，其他线程不再允许访问资源，直到某一线程释放信号量（这导致访问计数器加 1）为止。

下面是一个创建信号量对象的例子，使用语句：

```
CSenaphore Sema(2,2);
```

或

```
CSenaphore * sema = new CSemaphore(2,2);
```

其中的两个参数表示初始线程数和最大线程数。

下面，仍以上面的例子说明如何使用信号量对象进行线程同步。同样，只要对上面的例子程序中的线程控制函数稍作改动即可达到目的。

下面列出了文件 MySyn1View.cpp 改动后的代码：

```
// MySyn1View.cpp 文件的代码如下：
#include"afxmt.h"
……
//全局变量
int count = 0;
CSenaphore * sema = new CSemaphore(2,2);
……
UINT SynchroFunction(LPVOID param)
{
    int Data[5], i;
    CSingleLock singleLock(sema);
    singleLock.Lock();
    for(i = 0;i < 3;i ++ )
    {   count ++ ;
        Data[i] = count;
    }
    char str[50];
    str[0] = 0;
    for(i = 0;i < 3;i ++ )
    {   int len = strlen(str);
        wsprintf(&str[len],"%d",Data[i]);
    }
    ::Sleep(5000);          //延迟 5 s
    singleLock.Unlock();
    ::MessageBox((HWND)param,str,"Synchro Thread",MB_OK);
    return 0;
}
```

编译连接执行，可以看到，当选择“Thread”菜单下的“Synchronize”（同时启动 3 个同步线程）菜单项后，大约 5 秒钟，出现两个消息框，这表明两个线程同步地访问了受保护的资源。再过 5 秒

钟后，第二个消息框出现，表明它访问了资源。第二个线程之所以多花了5秒钟的时间才访问到资源，是因为头两个线程首先夺取了对资源的访问权，而信号量的设置只允许同时有两个线程可对资源进行访问，所以第三个线程不得不等待第一或第二个线程释放所占用的信号量。

## 习 题 十

1. 在图10.8中，编辑修改线程控制函数ThreadProc()，使每执行一次菜单命令Thread/Stop Thread，终止一个线程，即一个时钟停止运转。

2. 修改线程创建函数OnStartThread()和线程控制函数，使每执行一次菜单命令Thread/Start Thread，创建不同的线程，即各个线程有各自的控制函数。

3. 在图10.8中，编辑修改线程控制函数MyThread()，增加分针和时针，并与机器时钟一致。

4. 在题2的基础上，添加时钟其他数字等，并创建不同时区的时钟，如图10.1所示。

5. 类似于题2，在工程MySyn1中，修改OnSynchro()函数和同步线程控制函数，要求编写不同的同步线程控制函数，观察结果。

# 参 考 文 献


1 杨国兴. Visual C++ 6.0 实例教程. 北京：中国水利水电出版社，2001
2 王松. Visual C++ 6.0 程序设计与开发指南. 北京：高等教育出版社，2001
3 李鑫等. Visual C++ 6.0 编程基础与范例. 北京：电子工业出版社，2000
4 王华. Visual C++ 6.0 编程事例与技巧. 北京：机械工业出版社，1999
5 杨晓鹏等. Visual C++ 7.0 实用编程技术. 北京：中国水利水电出版社，2002
6 胡峪等. VC++ 高级编程技巧与示例. 西安：西安电子科技大学出版社，2001
7 灯芯工作室. Visual C++ 7.0 实战入门新概念. 北京：中国水利水电出版社，2001